Honeywell

FORTRAN IV

SERIES 16

SUBJECT:

Coding and Operating Procedures for Series 16 Fortran IV. Includes General Formats, Five Classifications of Statements, Subprograms, and a Fortran System Description.

SPECIAL INSTRUCTIONS:

This manual, BX32, Rev. 0, is a reprint of Order Number M-142. The order number has been changed to be consistent with the overall Honeywell publications numbering system. Contents are the same as the previous edition.

DATE:

April 1967

ORDER NUMBER:

BX32, Rev. 0 (Formerly M-142)

DOCUMENT NUMBER:

70130071364A

PREFACE

This manual describes Fortran coding and operating procedures for Series 16. Fortran IV processes the standard Fortran language specified by the United States American Standards Institute (USASI); it can express any problem of numerical computation.

Five classifications of Fortran statements (arithmetic and logical, control, input/output, specifications and subfunction) are discussed, along with general formats and subprograms.

Series 16 Fortran IV is a coded program designed to extend the power of Series 16 in the area of program preparation and maintenance. It is supported by comprehensive documentation and training; periodic program maintenance is furnished for the current version of the program, in accordance with established Honeywell specifications, provided it is not modified by the user.

File No.: 1A23

BX32

CONTENTS

INTRODUCTION

Every type of electronic computer is designed to respond to a special code, called a machine language, that differs for different types of computers. A program (set of instructions) telling a computer what steps to perform to solve a problem must ultimately be given to the computer in its own language. However, FORTRAN makes it unnecessary for the programmer to learn the machine language for a specific computer. Using FORTRAN, the programmer states in a relatively simple language, resembling familiar usage, the steps to be carried out by the computer. The program written in the FORTRAN language is entered into the computer and is automatically translated by the FORTRAN compiler into a program called the object program. The computer solves the problem by executing the object program which is expressed in a language the machine can understand.

Virtually any numerical or logical procedure can be expressed in the FORTRAN language. The FORTRAN system is intended to substantially reduce the time required to produce an efficient machine language program for the solution of a problem, and to relieve the programmer of a considerable amount of manual clerical work, minimizing the possibility of human error by relegating the mechanics of coding and optimization to the computer.

The name FORTRAN (FORmula TRANslator) was chosen because many of the statements resemble algebraic formulas. These statements define the computer operations and can be grouped into five classifications:

Arithmetic and logical
Control
Input/output
Specification
Subfunction

FORTRAN was originally written by IBM for the IBM 704 but has since been offered by several computer manufacturers. FORTRAN is relatively easy to learn. Of great importance is the fact that a FORTRAN source program can be compiled on many different computers, thereby avoiding the need for language translations when changing from one type of computer to another.

FORTRAN IV processes the standard FORTRAN language specified by the *United States American Standards Institute (USASI); it can express any problem of numerical computation. In particular, it deals easily with problems containing large sets of formulas and many variables, and permits variables to have up to three independent subscripts. For problems in which machine words have a logical rather than a numerical meaning, provisions in the FORTRAN language permit logical computation. In instances where FORTRAN may not be suited to the specific problem solution, provision has been made for calling machine language subroutines with FORTRAN statements. Certain statements in the FORTRAN language equip the object program with its necessary input and output programs. Those which deal with decimal information include conversion to and from binary and permit considerable freedom of format in the external medium. Arithmetic in the object program is generally performed with floating-point numbers. These numbers provide 23 binary digits (about 7 decimal digits) of precision and may have magnitudes between $10^{\pm 75}$ (DDP-24, -124, -224) or $10^{\pm 38}$ (DDP-116, -516). Full-word, fixed-point arithmetic, double precision floating-point arithmetic and complex number arithmetic is also provided.

Consider the quadratic equation:

$$3x^2 + 1.7x - 31.92 = 0$$

*The Proposed American Standard FORTRAN (see Appendix E) was followed in writing the FORTRAN compiler described in this manual.

The algebraic representation for one of the two roots of the equation could be written:

$$\text{ROOT} = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

where: A = +3
B = +1.7
C = −31.92

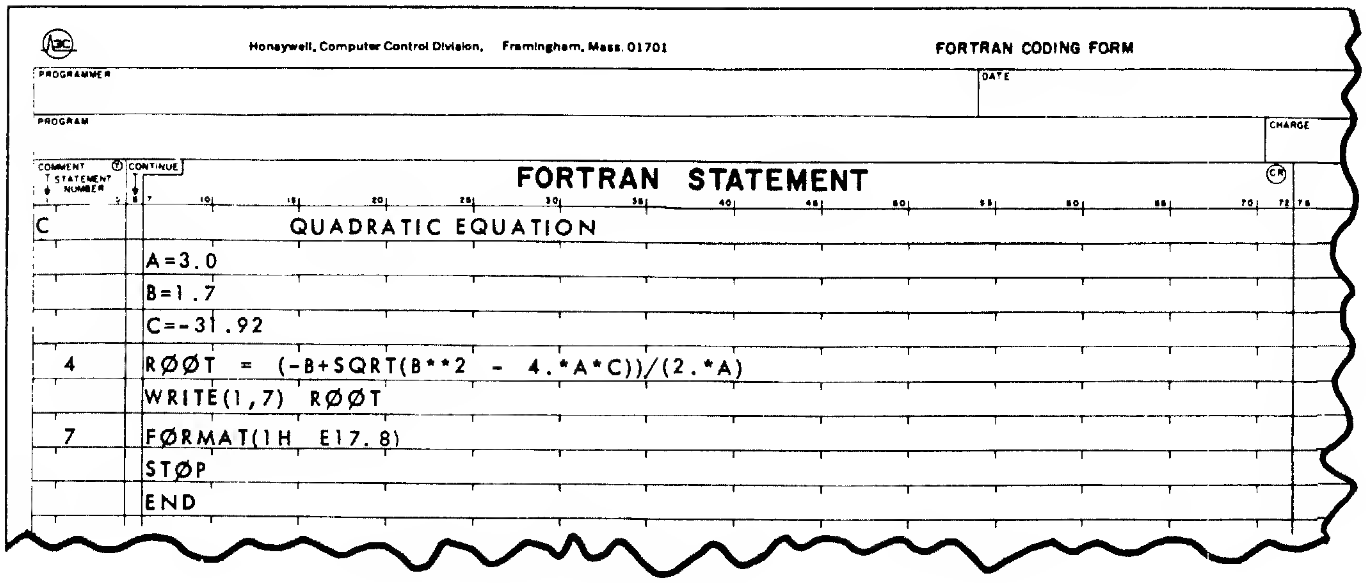
The complete FORTRAN program which describes this calculation and provides for output of the result may be written in eight separate statements, including the two statements which provide for the end of the job, as shown in Figure 1.

The first statement means: assign the floating-point value 3 to the variable A. The next two statements have a similar meaning. The fourth statement means: evaluate the expression on the right side, and assign the result to the variable ROOT. The fifth statement outputs the computed value of ROOT in a form indicated by the sixth statement. The last two statements indicate that the job is complete.

Notice the sequential nature of the program. The computer executes instructions in the same order as the order of the statements. For example, if the fourth statement were to be made the first statement, the computer would evaluate ROOT before obtaining the desired values of A, B, and C. ROOT would therefore be evaluated using arbitrary unknown values for these variables.

Figure 1 illustrates the use of variables. However, the same result could be obtained by writing statement number 4 as shown in Figure 2 in which the numerical values appear in the statement describing the evaluation of ROOT.

FORTRAN coding and operating procedures are given in this manual. In learning the FORTRAN system, the novice will profit by reading texts on the FORTRAN language, such as "A Guide to FORTRAN Programming" by D. A. McCracken (Wiley), "FORTRAN Autotester" by Smith and Johnson (Wiley), or "Comprehensive FORTRAN Programming" by James N. Haag (Hayden).



Honeywell, Computer Control Division, Framingham, Mass. 01701 FORTRAN CODING FORM

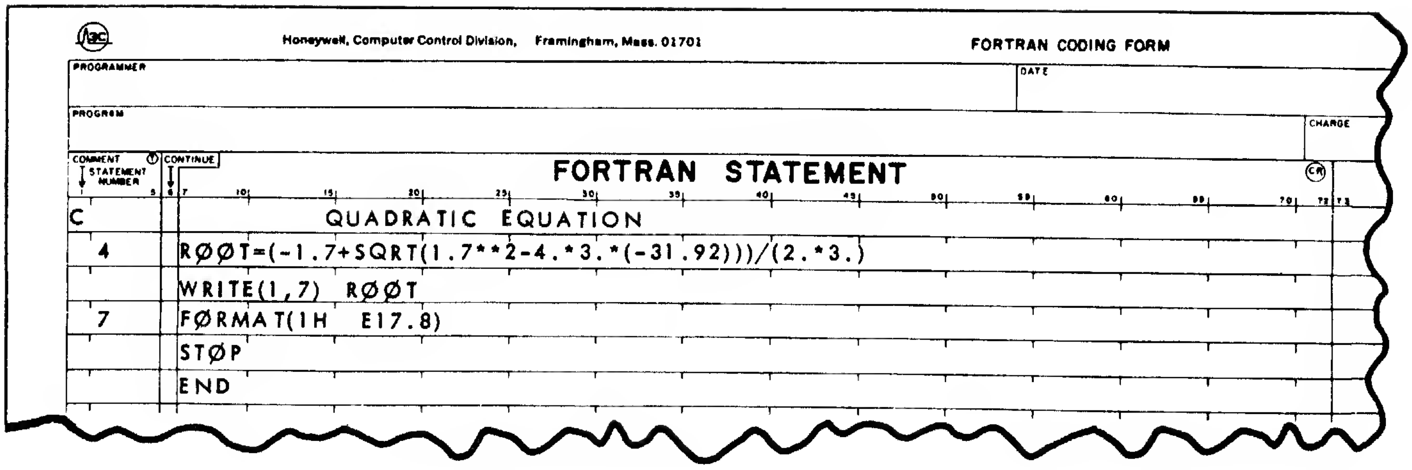
PROGRAMMER DATE

PROGRAM CHARGE

COMMENT / STATEMENT NUMBER — CONTINUE — FORTRAN STATEMENT

```
C          QUADRATIC EQUATION
      A=3.0
      B=1.7
      C=-31.92
    4 RØØT = (-B+SQRT(B**2 - 4.*A*C))/(2.*A)
      WRITE(1,7) RØØT
    7 FØRMAT(1H E17.8)
      STØP
      END
```

FIGURE 1



Honeywell, Computer Control Division, Framingham, Mass. 01701 — FORTRAN CODING FORM

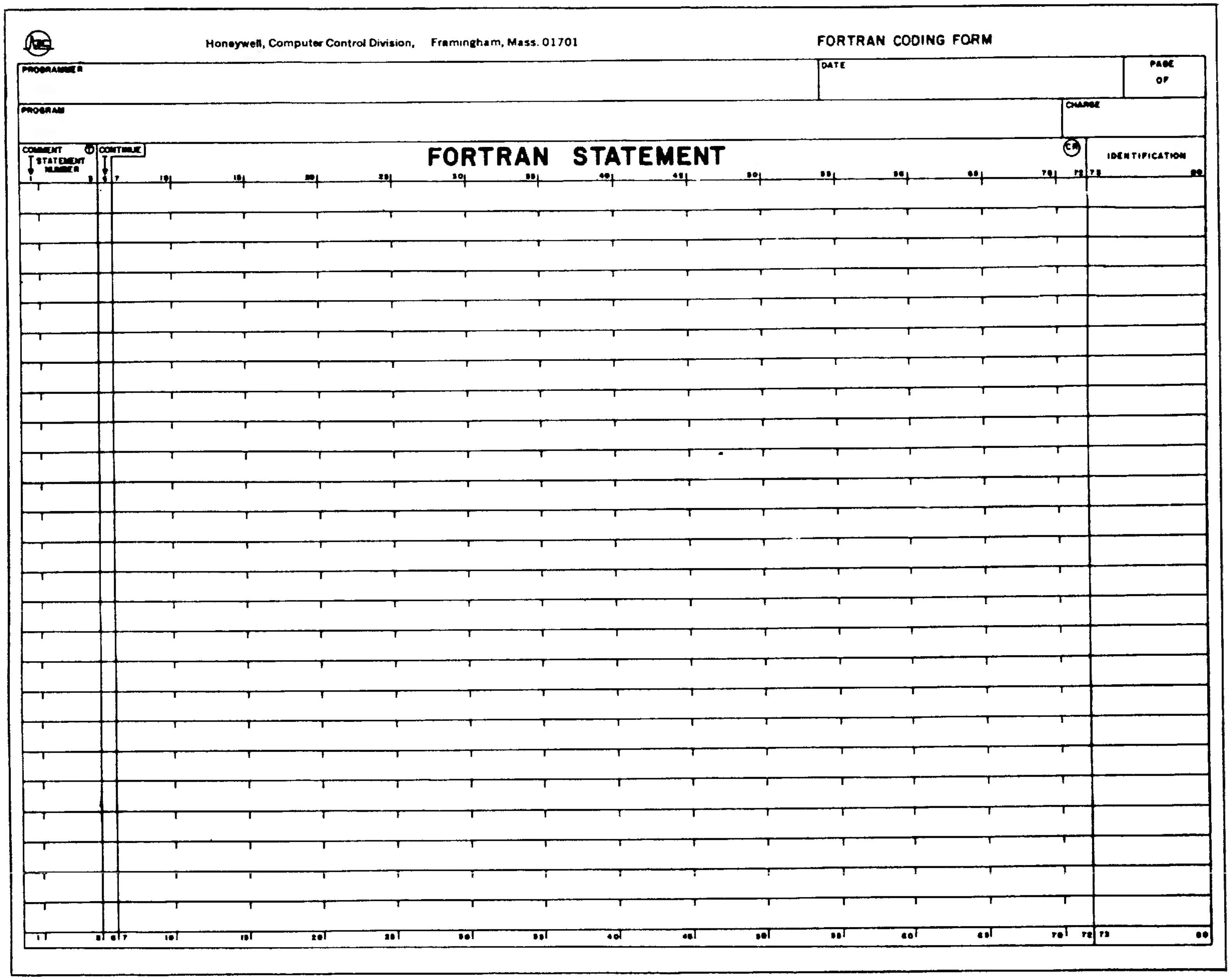
PROGRAMMER | DATE

PROGRAM | CHARGE

COMMENT / STATEMENT NUMBER | CONTINUE | FORTRAN STATEMENT

```
C               QUADRATIC EQUATION
   4  RØØT=(-1.7+SQRT(1.7**2-4.*3.*(-31.92)))/(2.*3.)
      WRITE(1,7) RØØT
   7  FØRMAT(1H  E17.8)
      STØP
      END
```

FIGURE 2



Honeywell, Computer Control Division, Framingham, Mass. 01701

FORTRAN CODING FORM

PROGRAMMER | DATE | PAGE OF

PROGRAM | CHARGE

COMMENT ↓ STATEMENT NUMBER | CONTINUE | FORTRAN STATEMENT | IDENTIFICATION

1 5 6 7 10 15 20 25 30 35 40 45 50 55 60 65 70 72 73 80

SECTION I
GENERAL FORMAT

FORTRAN CHARACTER SET

The FORTRAN language requires the use of the following characters, to specify constants, variable names, statement identifiers, or any other element of the language:

Letters

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

Digits

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Special

Period or decimal point (.), comma (,), plus (+), minus or hyphen (—), slash (/), asterisk (*), equals (=), open parenthesis ((), close parenthesis ()), apostrophe ('), dollar sign ($), and blank (or space)

LINE FORMAT

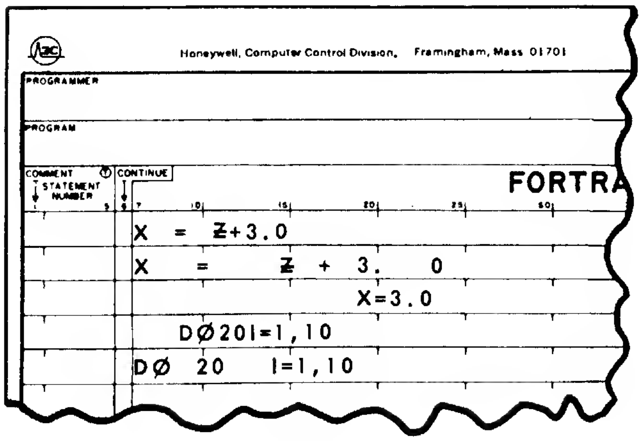
A line is made up of four fields: Statement Number Field, Line Continuation Field, Statement Field, and Identification Field. A coding form showing these fields appears at the end of the Introduction Section.

STATEMENT FIELD (Figure 1—1)

Any arithmetic statement, specification statement, control statement, I/O statement or function statement as described in later sections may appear in the statement field (card columns 7–72). Except in the Hollerith descriptor within a FORMAT statement (see I/O Section), blanks (or spaces) are ignored and may be used freely for appearance purposes.

LINE CONTINUATION FIELD (Figure 1—2)

If a FORTRAN statement is so large that it cannot conveniently fit into one statement field, the statement field of as many additional lines as is needed may be used to specify the complete statement. Continuation lines must have a character other than blank or zero in column 6. A statement may have any number of continuation statements with the following exceptions: (1) a DO statement must be on one line, and (2) the equals character (=) of an arithmetic statement must be on the first line if continuation lines are used. Any line which is not continued, or the first line of sequence of continued lines, must have a blank or the digit 0 in column 6.



Honeywell, Computer Control Division, Framingham, Mass 01701

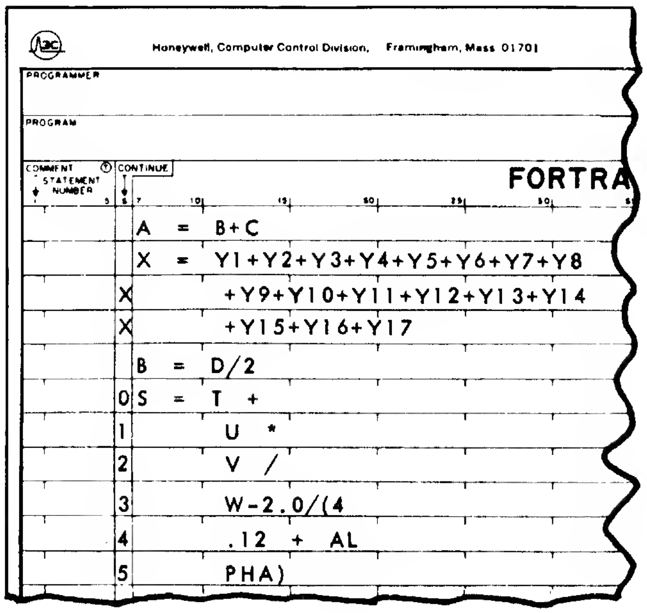
PROGRAMMER

PROGRAM

COMMENT STATEMENT NUMBER | CONTINUE | FORTRA

```
      X  =  Z+3.0
      X   =     Z  +  3.   0
                          X=3.0
           DØ20I=1,10
      DØ  20    I=1,10
```

FIGURE 1—1



Honeywell, Computer Control Division, Framingham, Mass 01701

PROGRAMMER

PROGRAM

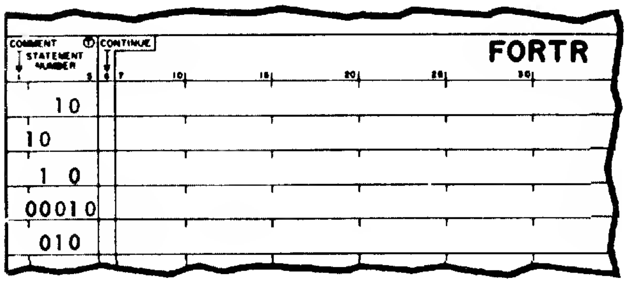
COMMENT STATEMENT NUMBER | CONTINUE | FORTRA

```
      A  =  B+C
      X  =  Y1+Y2+Y3+Y4+Y5+Y6+Y7+Y8
     X        +Y9+Y10+Y11+Y12+Y13+Y14
     X        +Y15+Y16+Y17
      B  =  D/2
     0S  =  T  +
     1        U  *
     2        V  /
     3        W-2.0/(4
     4        .12  +  AL
     5        PHA)
```

FIGURE 1—2

Statement Number Field (card columns 1–5)

If the statement located in the statement field is to be referenced by another statement (such as a GO TO statement), a reference number is placed in the statement number field (card columns 1–5). This reference number is made up of digits only. Leading zeros and all blanks in this field are ignored. For example, the reference numbers shown below are all of the same value.



COMMENT
STATEMENT NUMBER
CONTINUE
FORTR

```
   10
10
  1 0
00010
  010
```

Specification statements (Section IV), FUNCTIONS, SUBROUTINE, or END statements, or arithmetic statement functions should not have statement numbers. All other statements may have a number whether or not that statement is referred to by another statement. Statement numbers need not be in sequence. (See Figure 1–3.)

Identification Field

The last eight columns of a line (card columns 73–80) are reserved for sequence identification. This field is ignored by the compiler and may be left blank if desired.

Comment Lines

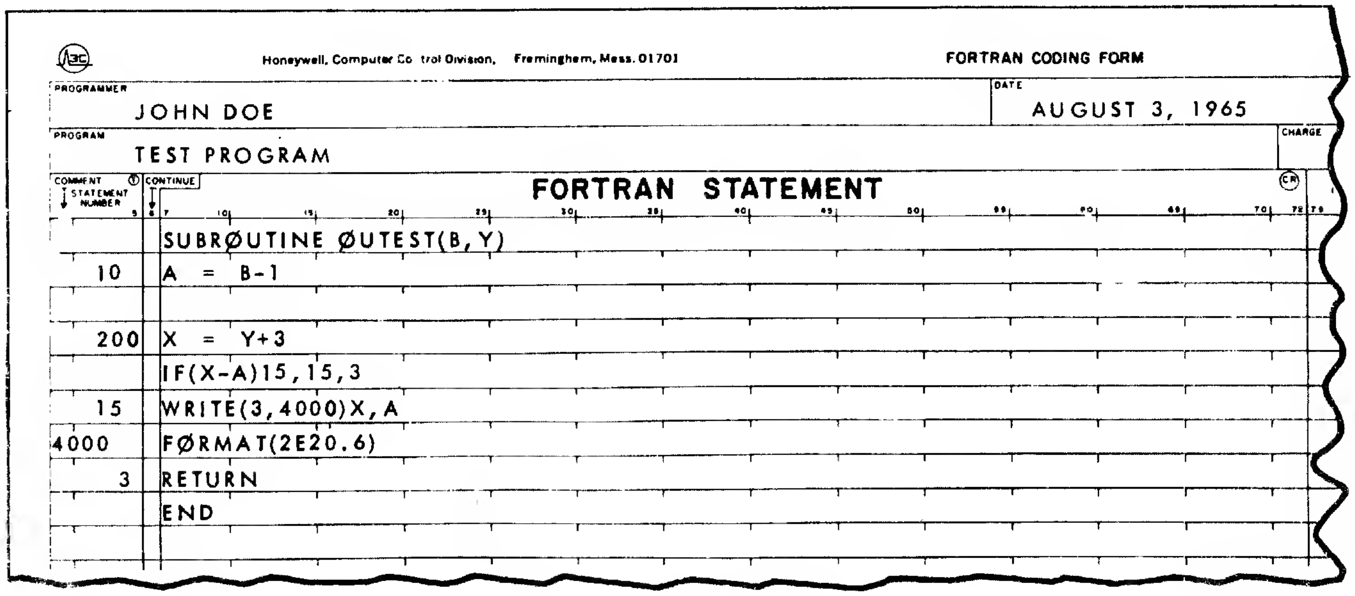
Any line which starts with the letter "C" in column 1 is presumed to be a line of comments. This line is printed onto any listings requested but is otherwise ignored by the compiler. The statement number, line continuation, and statement fields may be used in any format for comment purposes. (See Figure 1–4.) The identification field, however, is used for sequencing only. A blank line is ignored.

Special Control Lines

A line that starts with a dollar sign character ($) in column 1 is presumed to be a special control line. The digit in column 2 indicates the type of control, and columns 3–72 are ignored and may be used for comments. Columns 73–80 are used for card sequencing purposes. The following codes are currently used as shown in Figure 1–4.

$0	End of job
$1	Continuation segment of a chain job
$2–$9	Spares

The use of the control lines is explained in later sections.



Honeywell, Computer Co trol Division, Framingham, Mass. 01701
FORTRAN CODING FORM
PROGRAMMER
DATE
PROGRAM
CHARGE
COMMENT
STATEMENT NUMBER
CONTINUE
FORTRAN STATEMENT

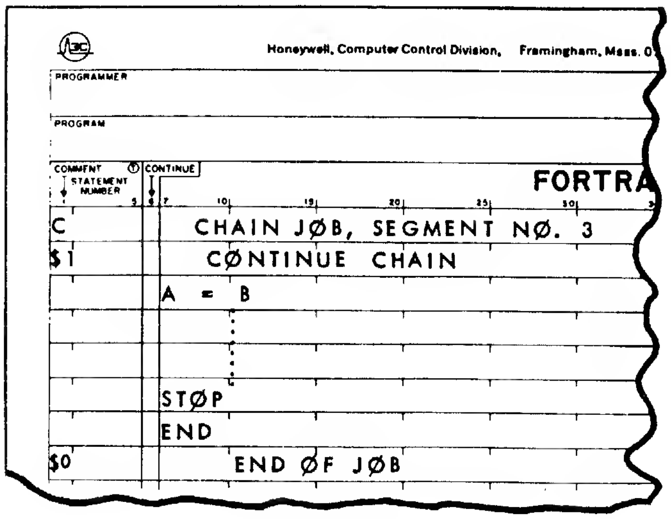
PROGRAMMER: JOHN DOE
DATE: AUGUST 3, 1965
PROGRAM: TEST PROGRAM

```
      SUBRØUTINE ØUTEST(B,Y)
   10 A = B-1

  200 X = Y+3
      IF(X-A)15,15,3
   15 WRITE(3,4000)X,A
 4000 FØRMAT(2E20.6)
    3 RETURN
      END
```

FIGURE 1—3

CONSTANTS AND VARIABLES

Any programming language must provide a means of expressing numerical constants and variable quantities. FORTRAN IV allows six types of constants and five types of variables to be expressed in an arithmetic expression: integer, real, double precision, complex, logical, and Hollerith.


Honeywell, Computer Control Division, Framingham, Mass.
PROGRAMMER
PROGRAM
COMMENT
STATEMENT NUMBER
CONTINUE
FORTRA

```
C     CHAIN JØB, SEGMENT NØ. 3
$1     CØNTINUE CHAIN
      A = B
          .
          .
          .
      STØP
      END
$0        END ØF JØB
```

FIGURE 1—4

Constants

Constants may have a sign and must start with a digit or a decimal point following one of the formats defined in Table 1–1.

Variables

Variable names must start with a letter, and contain one to six characters made up of letters and digits only. The mode of the variable and the number of memory words reserved for it are defined in Table 1–2.

SUBSCRIPT NOTATION

If programming were done using only the types of variables presented in the preceding pages, laborious programming would be necessary to carry out relatively simple iterative calculations or logical steps such as those encountered in the addition of two vectors or the selection of a specific entry in a list of numbers. However, it is possible to employ the subscript notation of mathematics to simplify the programming of such problems.

A mathematician would denote that c_i is the sum of the vectors (a_1, a_2, a_3) and (b_1, b_2, b_3) as follows:

$$c_i = a_i + b_i \qquad i = 1, 2, 3$$

Note that the first part of the statement ($c_i = a_i + b_i$) is a general statement which becomes, in effect, three specific statements:

$$c_1 = a_1 + b_1$$
$$c_2 = a_2 + b_2$$
$$c_3 = a_3 + b_3$$

when the values 1, 2, and 3 are assigned to i.

By using the FORTRAN language, it is possible to make general statements like $c_i = a_i + b_i$, and to make other statements which assign the desired values to i. When a general statement is executed, it is always executed in one of its specific senses. For example, if the variable I has the value 3 when the FORTRAN equivalent of $c_i = a_i + b_i$

C(I) = A(I) + B(I)

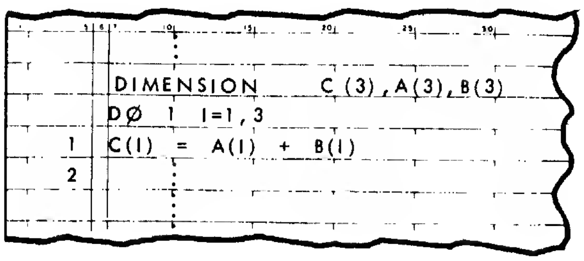
is executed, the values denoted by A(3) and B(3) are added and the sum is assigned as the value of C(3). Thus, to compute the sum vector

(C(1), C(2), C(3))

it is necessary to execute the general statement three times, with I being equal to 1 the first time, 2 the second time, and 3 the third time. Therefore, in addition to providing for arithmetic statements with subscripted variables, it is necessary to provide a method of stating that a given set of such statements should be executed repetitively for certain values of the subscript. The FORTRAN statement which provides this ability is called a DO statement. An example of a DO statement is

DO 20 I = 1, 250

This statement instructs the computer to execute all statements which immediately follow, up to and including the statement numbered 20, 250 times (the first time for I = 1, the second time for I = 2, and so on, and the last time for I = 250), and then go on to the statement following statement 20. Thus, to return to the example of vector addition, the FORTRAN statements necessary to add A(I) and B(I) are:



```
          .
      DIMENSIØN    C(3),A(3),B(3)
      DØ 1 I=1,3
    1 C(I) = A(I) + B(I)
    2     .
          .
```

TABLE 1—1.
CONSTANT FORMAT

MODE OF CONSTANT	GENERAL FORMAT	EXAMPLES
INTEGER (Occupies 1 word)	1 to 7 decimal digits (only 5 digits for DDP-116, -516). A preceding + or — sign is optional. The magnitude of the constant must be less than 8388608 (32768 for DDP-116). No decimal point is allowed.	3 +1 —28987 0
REAL (Single precision floating point — occupies 2 words)	Any number of decimal digits may be used but only the most significant 7 digits will be retained.	17. 5.0 —.0003
	The requirement of a decimal point in constants having a decimal exponent is also optional. A preceding + or — sign is optional.	.0000005 0.0
	A decimal exponent with magnitude less than 76 (38 for DDP-116, -516) and preceded by an E may follow the constant, but is optional.	5.0E—7 +5E—7 5.0E3
	A decimal point must be included at the beginning, at the end, or within the constant.	.5E+3 5000.
DOUBLE PRECISION (Double precision floating point — occupies 3 words)	Any number of decimal digits may be used (followed by a decimal exponent) but only the most significant 13.7 (11.5 for DDP-116, -516) digits will be retained.	1.736247D5 3002625D—4 1DO 1234.567890123DO
	A decimal exponent with a magnitude less than 76 (38 for DDP-116, -516) and preceded by a D must follow the constant. A preceding + or — sign is optional. A decimal point is optional and may be included either at the beginning, at the end, or within the constant.	
COMPLEX (Two single precision floating point constants — occupies 4 words)	Two REAL format constants separated by a comma and enclosed by parentheses. The first constant represents the real part of the complex constant and the second constant represents the imaginary part. Either constant may be preceded by an optional + or — sign.	(17.0,5.0) (—2.35,1.E—6)
LOGICAL (Occupies 1 word)	The word TRUE or FALSE preceded and followed by a decimal point. TRUE generates an INTEGER format 1 and FALSE generates an integer FORMAT 0.	.TRUE. .FALSE
HOLLERITH (Occupies 2 words)	An integer constant, followed by the letter H, followed by the integer constant number of characters (blanks included) equal to the integer constant.	1HX 4HB.24 4HTEST 4H Z
	Any acceptable FORTRAN character including blank may be used as an integer constant and can be up to 4 characters long (2 characters for DDP-116, -516). This type of constant is only accepted in CALL or DATA statements.	

When the statement numbered 2 is encountered, the values of C(1), C(2), and C(3) will have been

GENERAL FORM	EXAMPLES

A real, integer, double precision, complex, or logical variable followed by parentheses enclosing 1, 2, or 3 subscript expressions separated by commas. The subscripts must be integer and one of the following formats:
k
v
v±k
c*v
c*v±k
k or c = integer constants
v = integer variable
	A(I)
	K(3)
	BETA (5*J — 2,K+2,L)
	X(I,J,K)
	X(I+4,6,2*I)

computed and stored. (The DO statement is discussed in detail in Section III.)

For each variable that appears in subscripted form, the size of the array (i.e., the maximum values which its subscripts can attain) must be stated in a DIMENSION statement preceding the first executable statement in the source program. (DIMENSION statements are described in Section IV.) The value of a subscript expression must not be greater than the corresponding array dimension.

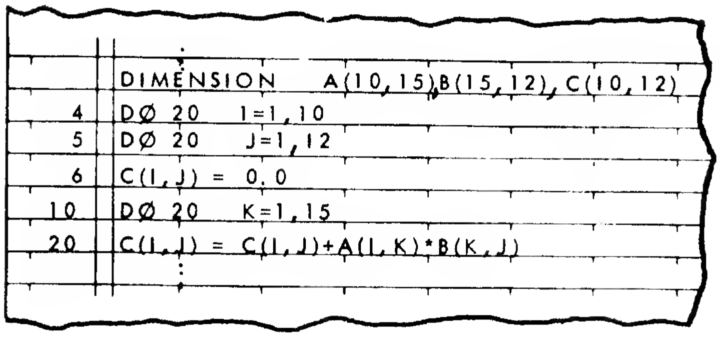
Subscripting to handle two- and three-dimensional arrays greatly facilitates the solving of many engineering and scientific problems which require matrix manipulations for their solution. The following example of matrix multiplication illustrates DO nests and multiple subscripts. (A DO nest is a set of two or more DO statements, the range of one of which includes the ranges of the others.)

EXAMPLE OF MATRIX MULTIPLICATION

Problem. Given matrix A with dimensions 10 x 15, and matrix B with dimensions 15 x 12, compute the elements of C_{ij} of matrix C = AB.

Solution. To compute any element C_{ij}, select the i row of A and the j column of B, and sum the products of their corresponding elements. The general formula for this computation is:

$$C_{ij} = \sum_{k=1}^{k=15} A_{ik} B_{kj}$$



```
        .
        DIMENSION    A(10,15),B(15,12),C(10,12)
    4   DØ 20    I=1,10
    5   DØ 20    J=1,12
    6   C(I,J) = 0.0
   10   DØ 20    K=1,15
   20   C(I,J) = C(I,J)+A(I,K)*B(K,J)
        .
        .
```

Statement 5 indicates that the program is to be repeated 12 times, first for J = 1, then J = 2, ...

TABLE 1—2.

VARIABLE FORMAT

MODE OF VARIABLE	GENERAL FORMAT	EXAMPLES
INTEGER (1 word reserved)	1 to 6 alphabetic or numeric characters, the first of which is the letter I, J, K, L, M, or N. If the first character is not one of these letters, it is an integer variable only if also mentioned in an INTEGER declaration.	I2 NAME NUMBER KEY J
REAL (2 words reserved)	1 to 6 alphabetic or numeric characters, the first of which is not the letter I, J, K, L, M or N. If the first character is one of these letters, it is a real variable only if mentioned previously in a REAL declaration.	VAR43 X TEST DELTA
DOUBLE PRECISION (3 words reserved)	1 to 6 alphabetic or numeric characters, the first of which is a letter and the variable is mentioned previously in a DOUBLE PRECISION declaration.	ALPHA K33 Z
COMPLEX (4 words reserved)	1 to 6 alphabetic or numeric characters, the first of which is a letter and the variable is mentioned previously in a COMPLEX declaration.	J6 XTEST XFL4
LOGICAL (1 word reserved)	1 to 6 alphabetic or numeric characters, the first of which is a letter and the variable is mentioned previously in a LOGICAL declaration.	LOGT LTEST A23L

J = 12. Notice that for each repetition of statements 6 through 20, statement 20 is executed 15 times (first for K = 1, then for K = 2, and so on). Thus, when the process demanded by statement 5 is complete, the I row of the product matrix has been computed and stored. Control then returns to statement 4 to obtain a new value for I, and statements 5 through 20 are repeated for the new value. The process continues until all of the rows of the product matrix are produced.

Arrangement of Arrays in Storage

In the object program, a two-dimensional array A will be stored sequentially in the order $A_{1,1}$, $A_{2,1}$, . . . $A_{m,1}$; $A_{1,2}$, $A_{2,2}$, . . . , $A_{m,2}$; . . . , $A_{m,n}$. Thus the array is stored with the first of its subscripts varying most rapidly, and the last varying least rapidly. The same is true of the subscripts of three-dimensional arrays.

All arrays are stored forward in storage; i.e., the following sequence is in the order of increasing absolute locations.

ARRAY			ARRANGEMENT IN STORAGE	
$A_{1,1}$	$A_{2,1}$	$A_{3,1}$	$A_{1,1}$	(1)
$A_{1,2}$	$A_{2,2}$	$A_{3,2}$	$A_{2,1}$	(2)
$A_{1,3}$	$A_{2,3}$	$A_{3,3}$	$A_{3,1}$	(3)
			$A_{1,2}$	(4)
			$A_{2,2}$	(5)
			$A_{3,2}$	(6)
			$A_{1,3}$	(7)
			$A_{2,3}$	(8)
			$A_{3,3}$	(9)

SECTION II

ARITHMETIC AND LOGICAL STATEMENTS

FORMAT

An arithmetic statement (or arithmetic formula) defines a numerical calculation. A FORTRAN arithmetic statement resembles very closely a conventional arithmetic formula. It consists of a single variable to be computed, followed by an equals sign (=), followed by an arithmetic expression.

GENERAL FORM	EXAMPLES

a = b where a is a variable (subscripted or not subscripted) and b is an expression.	L1 = K + 1
	A(1) = B(1) + SIN (C(1))

In a FORTRAN arithmetic statement, however, the equals sign means *is to be replaced by* rather than *is equivalent to.* An arithmetic statement instructs the computer to compute the value of the expression on the right side of the equals sign and to store the result in the memory location assigned to the variable on the left side of the equals sign. Thus the arithmetic statement Y = A − B*C directs the computer to replace the value of the variable Y with the computed value of the expression A − B*C.

The expression on the right side of an arithmetic statement may be any sequence of constants, variables (subscripted or not subscripted), and functions, separated by operation symbols, commas, and parentheses to form a meaningful expression.

The five basic operations in the FORTRAN language are:

OPERATOR	DEFINITION	EXAMPLE
+	Add	a + b
−	Subtract	a − b
*	Multiply	a * b
/	Divide	a / b
**	To the exponent	a ** b

One arithmetic operation symbol cannot immediately follow another. An arithmetic expression must not contain logical data.

Following are examples of arithmetic statements:

FORMULA	DESCRIPTION
A = B	Store the value of B in A.
I = B	Truncate B to an integer and store in I.
A = I	Convert I to floating point, and store in A.
I = I + 1	Add 1 to I and store in I. (This illustrates that an arithmetic formula is not an equation but rather a command to replace a value.)
A = 3.0*B	Replace A with 3B.

Parentheses are used as in ordinary mathematical notation to specify order. For example, $(A(B + C))^D$ written in FORTRAN is (A*(B+C))**D. Some exceptions to the rules of ordinary mathematical notation are:

a. In ordinary notation AB means A times B or A * B. However, AB does not mean A*B in FORTRAN. The multiplication symbol cannot be omitted.

b. In ordinary usage, expressions such as A/B*C are considered ambiguous. However, such expressions are allowed in FORTRAN and are interpreted as follows:

A/B*C *means* (A/B)*C
A*B/C *means* (A*B)/C
A/B/C *means* (A/B)/C

Thus, for example, A/B/C*D*E/F *means* ((((A/B)/C)*D)*E)/F. That is, the order of operations is taken from left to right.

c. The expression A^{B^C} is often considered meaningful. However, in FORTRAN the corresponding expression (A**B**C) is not allowed. It should be written (A**B)**C if $(A^B)^C$ is meant, or A**(B**C) if $A^{(B^C)}$ is intended.

d. The expression A/−B, which in ordinary notation would be interpreted as A divided by the negative value of B, is illegal in FORTRAN since two operators are not permitted to be side-by-side. It should be written as A/(−B) or −A/B.

OPERATOR HIERARCHY

When a group of operators are not enclosed in parentheses, the following implied hierarchy dictates the order in which the operations are performed:

PRIORITY	OPERATOR	OPERATION
8	FUNCTIONS	Function subroutine
7	**	Exponentiation
6	*,/	Multiply or divide
5	+,—	Add or subtract
4	.LT.,.LE.,.EQ., .NE.,.GT.,.GE.	Relational operators
3	.NOT.	Logical negate
2	.AND.	Logical AND
1	.OR.	Logical OR

The operator with the highest priority takes precedence over the other operators in the same expression.

Thus, when the arithmetic statement

F = A+SIN(B)*C**D

is encountered at execution time, the computer would perform the indicated operations in the following order:

a. SIN(B)

b. C**D

c. (SIN(B))*(C**D)

d. A+((SIN(B))*(C**D))

Again, parentheses that have been omitted from a sequence of consecutive multiplications and divisions are interpreted as being grouped from the left. Thus: A/B/C/D/E *means* (((A/B)/C)/D)/E)

NOTE:—Special care must be taken to indicate the order of integer multiplication and division. FORTRAN integer arithmetic is "greatest integer" arithmetic; i.e., truncated or remainderless. Therefore, the expression 5*4/2 is evaluated as 10, but 5/2*4 is evaluated as 8. To insure the desired result in integer multiplication and division, it is suggested that parentheses be used in the expression.

MIXED EXPRESSIONS

In general, FORTRAN IV does not allow variables or constants of one mode to appear in the same statement with variables or constants of another mode. For example, integer variables cannot be added to real variables, real variables cannot be added to logical variables, etc.

There are, however, the following exceptions to this general rule:

a. Arguments of functions or subroutines may be any mode (or Hollerith) but must agree in mode in transmission.

```
CALL DUMP (A,I,CPLX,LOGI)
ACPLX = BCPLX + CABS (REAL)
```

b. Subscripts of subscripted variables must be expressed in integer mode.

A = ARRAY (I) + TABLE (4,J,2)

c. When raising a value to a power, the power may have a different mode than that of the value being raised. Only those modes shown in the following eight examples are acceptable where I, R, D and C indicate INTEGER, REAL, DOUBLE, or COMPLEX mode variables or constants.

*I**I*	INTEGER Result
*R**I*, *R**R*	REAL Result
*R**D*, *D**I*, *D**R*, *D**D*	DOUBLE Result
*C**I*	COMPLEX Result

d. Arguments may be intermixed between REAL, DOUBLE, or COMPLEX modes. If one of the arguments of an arithmetic operation (+,—,*,/) is REAL and the other is DOUBLE or COMPLEX, the REAL argument is converted to DOUBLE or COMPLEX format, and the result of the arithmetic operation is in DOUBLE or COMPLEX format.

e. Automatic mode conversion will take place "across" the equals sign of an arithmetic expression between INTEGER, REAL, or DOUBLE format. The following list shows allowable modes of the variable preceding the equals sign and the expression following the equals sign.

```
I = I,  I = R,  I = D
R = I,  R = R,  R = D
D = I,  D = R,  D = D
C = C
L = L
```

f. All alphanumeric data occupies one word per variable and is typed as INTEGER mode.

LOGICAL EXPRESSION

The result of any logical expression is a TRUE or a FALSE logical quantity. Two types of operators may be used in logical expressions: relational operators or logical operators.

RELATIONAL OPERATORS

Relational operators join two arithmetic expressions. The mode of the two arithmetic expressions must be either INTEGER, REAL, DOUBLE PRE-

CISION, or a combination of REAL and DOUBLE PRECISION.

The result of a relational operation is a TRUE or FALSE answer to the question posed by the relational operator.

RELATIONAL OPERATOR	MEANING	EXAMPLES
.LT.	Less than	A.LT.B
.LE.	Less than or equal to	X.LE.Z+ DELTA
.EQ.	Equal to	(I+3)/5 .EQ.10
.NE.	Not equal to	R.NE.0.4
.GT.	Greater than	SIN(C) .GT.3.14157
.GE.	Greater than or equal to	D.GE.DF

Logical Operators

Logical operators generate a TRUE or FALSE answer based on other TRUE or FALSE logical quantities. These logical quantities may be logical constants, logical variables or subscripted variables, logical functions, relational expressions (described in the previous paragraph), or other logical expressions (enclosed in parentheses).

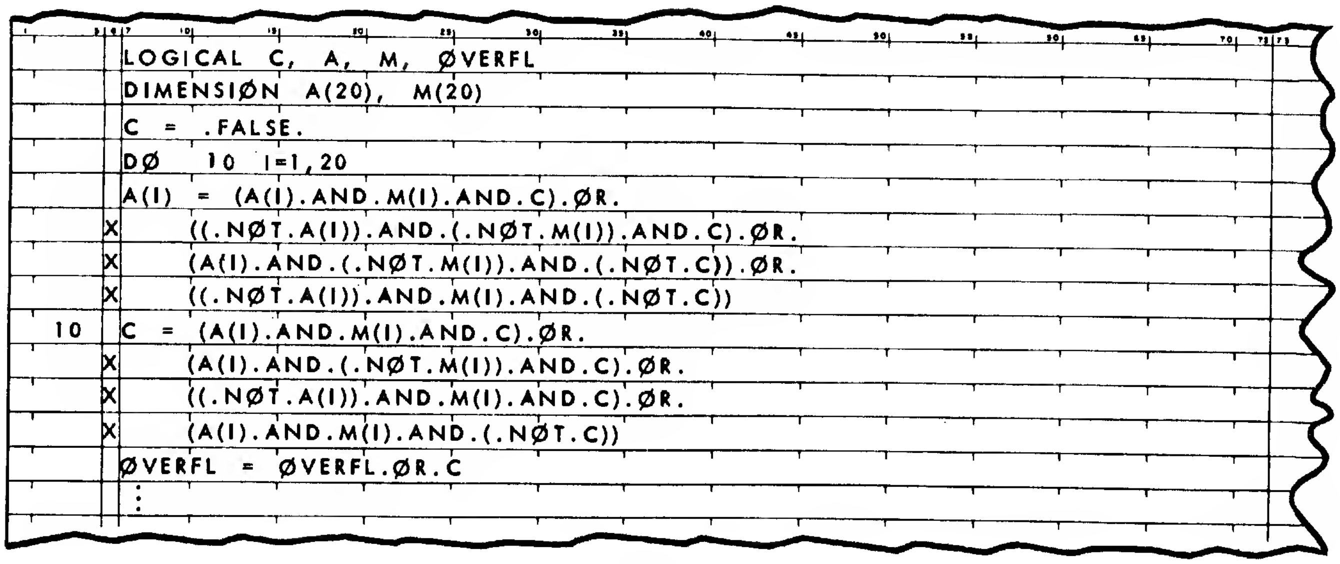
Example of Logical Statement

The logic of a simple adder is to be simulated in FORTRAN language. (See Figure 2–1.) A 20-bit accumulator word and a 20-bit memory word are to be added with a 20-bit result appearing in the accumulator.

LOGICAL OPERATOR	MEANING	EXAMPLES
.NOT.	Reverse the state of the logical quantity that follows.	.NOT.L .NOT.X.GE.6.47

.AND.	Generate a logical result based on two logical quantities as follows:	L1.AND.L2
		L1.AND.(.NOT.L2)
		(X.LT.E).AND.L3
		L3.AND.(Y.NE.0.0)

		A B A.AND.B
	T.AND.T ⟶T	1 1 1
	T.AND.F ⟶F	1 0 0
	F.AND.T ⟶F	0 1 0
	F.AND.F ⟶F	0 0 0
.OR.	Generate a logical result based on two logical quantities as follows:	L1.OR.L2 (.NOT.L1).OR.L2 (L1.AND.L2).OR.(L2.AND.L3) L3.OR(Y.NE.X) ((L1.OR.L2).AND.L3).OR.L4
		A B A.OR.B
	T.OR.T ⟶T	1 1 1
	T.OR.F ⟶T	1 0 1
	F.OR.T ⟶T	0 1 1
	F.OR.F ⟶F	0 0 0



```
      LOGICAL C, A, M, ØVERFL
      DIMENSIØN A(20), M(20)
      C = .FALSE.
      DØ 10 I=1,20
      A(I) = (A(I).AND.M(I).AND.C).ØR.
     X     ((.NØT.A(I)).AND.(.NØT.M(I)).AND.C).ØR.
     X     (A(I).AND.(.NØT.M(I)).AND.(.NØT.C)).ØR.
     X     ((.NØT.A(I)).AND.M(I).AND.(.NØT.C))
   10 C = (A(I).AND.M(I).AND.C).ØR.
     X     (A(I).AND.(.NØT.M(I)).AND.C).ØR.
     X     ((.NØT.A(I)).AND.M(I).AND.C).ØR.
     X     (A(I).AND.M(I).AND.(.NØT.C))
      ØVERFL = ØVERFL.ØR.C
       :
```

FIGURE 2—1

SECTION III
CONTROL STATEMENTS

GO TO STATEMENTS

There are three types of GO TO statements:
- Unconditional GO TO
- Assigned GO TO
- Computed GO TO

Unconditional GO TO Statement

The unconditional GO TO statement is in the form:

GO TO k

where k is a statement number. Execution of this statement causes the statement identified by the statement number to be executed next.

ASSIGN Statement

This statement is required with the assigned GO TO and is in the form:

ASSIGN k TO i

where k is a statement number and i is an integer variable name. After execution of such a statement, subsequent execution of any assigned GO TO statement using the integer variable (i) will cause the statement identified by the assigned statement number to be executed next.

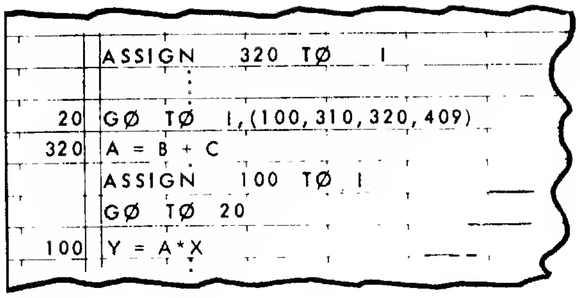
Assigned GO TO Statement

This type of GO TO takes the form:

GO TO i, $(k_1, k_2, \ldots, k_n)$

where i is an integer variable reference, and the k's are statement numbers.

At the time of execution of an assigned GO TO statement, the current value of i must have been assigned by the previous execution of an ASSIGN statement to be one of the k_n statement numbers in the parenthesized list. The execution of such an assigned GO TO statement causes the statement identified by the statement number k to be executed next.



```
      ASSIGN   320 TØ  I
         :
   20 GØ  TØ  I,(100,310,320,409)
  320 A = B + C
      ASSIGN   100 TØ  I
      GØ  TØ  20
  100 Y = A*X
         :
```

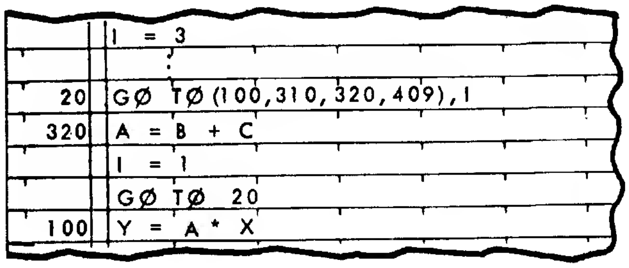
Computed GO TO Statement

This type of GO TO statement is an n-way branch where the value of an integer variable determines the statement number to which the transfer is effected. The format of a computed GO TO statement is:

GO TO $(k_1, k_2, \ldots, k_n)$, i

where the k's are statement numbers and i is an integer variable reference.

Execution of the above statement causes the statement identified by the statement label k_j to be executed next, where j is the value of i at the time of the execution. This statement is defined only for values such that $1 \leq j \leq n$. In other words, if $j = 3$, the statement identified by the third statement number in the list will be executed next.



```
      I = 3
        :
   20 GØ TØ (100,310,320,409),I
  320 A = B + C
      I = 1
      GØ TØ  20
  100 Y = A * X
```

IF STATEMENTS

All GO TO statements cause an unconditional transfer to a single statement. The IF statements branch to one of two or three statements depending on a condition. Two types of IF statements are recognized:

Arithmetic IF statement
Logical IF statement

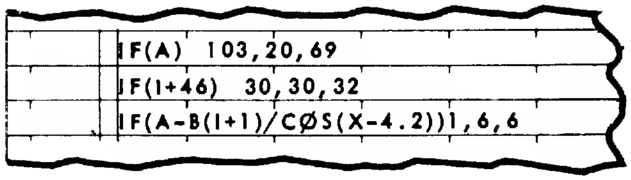
Arithmetic IF Statement

This type of statement takes the form:

IF (e) k_1, k_2, k_3

where e is any arithmetic expression of type integer, real, or double precision, and the k's are statement labels.

The arithmetic IF is a three-way branch in which the expression e is evaluated. If the expression e is negative, there is a transfer to the statement identified by k_1; if the expression is zero, there is a transfer to statement k_2; if the expression is positive (non-zero), there is a transfer to k_3.



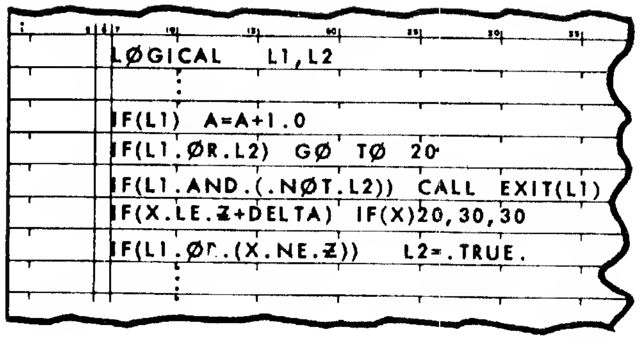
```
      IF(A) 103,20,69
      IF(I+46) 30,30,32
      IF(A-B(I+1)/CØS(X-4.2))1,6,6
```

As can be seen in the second and third IF statement above, the three-way hranch can be converted to a two-way branch by assigning the same statement number to two of the three statement numbers in the list.

Logical IF Statement

The logical IF statement takes the general form:

IF (e) S



```
      LØGICAL   L1,L2
         :
      IF(L1) A=A+1.0
      IF(L1.ØR.L2) GØ TØ 20
      IF(L1.AND.(.NØT.L2)) CALL EXIT(L1)
      IF(X.LE.Ƶ+DELTA) IF(X)20,30,30
      IF(L1.ØR.(X.NE.Ƶ))  L2=.TRUE.
         :
```

where e is a logical expression and S is any executable statement except a DO statement or another logical IF statement. Upon execution of this statement, the logical expression e is evaluated. If the value of e is FALSE, statement S is ignored and the next sequential statement is executed. If the value of e is TRUE, statement S is executed followed by the next sequential statement (unless statement S is a GO TO or arithmetic IF statement).

Computer Test IF Statements

FORTRAN II statements such as IF ACCUMULATOR OVERFLOW, IF (SENSE LIGHT m), IF (SENSE SWITCH m), IF QUOTIENT OVERFLOW, IF DIVIDE CHECK, and SENSE LIGHT are replaced in FORTRAN IV with functions on the library tape that provide the same tests and action. (See Table 3—1.) These functions are explained in detail in their respective program listings and include the following:

SLITE(I). Sense light I will be set. All sense lights will be reset if I = 0. (DDP-24, -124, and -224 have 24 simulated sense lights. DDP-116, -516 have 16 simulated sense lights.)

SLITET(I,J). Sense light I will be turned off. J is set to 1 if sense light I was set, or set to 2 if sense light I was reset.

SSWTCH (I,J). J is set to 1 if sense switch I is set, or set to 2 if sense switch I is reset. (DDP-24, -124, and -224 have six sense switches; DDP-116, -516 have four sense switches.)

TABLE 3—1.

FORTRAN II VERSUS FORTRAN IV STATEMENTS

FORTRAN II	FORTRAN IV
SENSE LIGHT 3	CALL SLITE (3)
IF (SENSE LIGHT 6) 20, 30	CALL SLITET (6,I) GO TO (20,30),I
IF (SENSE SWITCH 1) 15, 13	CALL SSWTCH (1,N) GO TO (15,13),N
IF ACCUMULATOR OVERFLOW 5, 10	CALL OVERFL(K) GO TO (5,10),K
IF QUOTIENT OVERFLOW 5, 10	CALL OVERFL(K) GO TO (5,10),K
IF DIVIDE CHECK 5, 10	CALL OVERFL(K) GO TO (5,10),K

OVERFL(J). The error flag is turned off. J is set to 1 if the error flag was on, or set to 2 if the error flag was off. The error flag is located in subroutine F$ER and is set on anytime an error condition (including underflow as well as overflow) exists in an arithmetic calculation or in an input/output conversion.

DO STATEMENT

The general format for the DO statement is:

DO n i = m_1, m_2, m_3
or
DO n i = m_1, m_2

where:

1. n is the statement number of an executable statement. This statement, called the terminal statement of the associated DO, must physically follow and be in the same program as that DO statement. The terminal statement may not be a GO TO (of any form), arithmetic IF, RETURN, STOP, PAUSE or DO statement, nor a logical IF containing any of these forms.

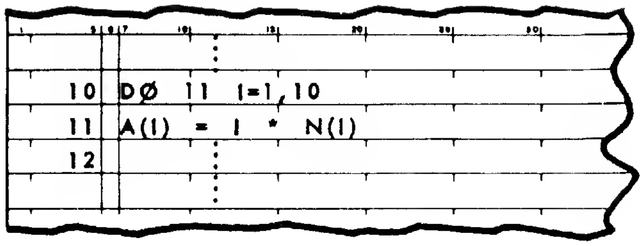
2. i is an integer variable.

3. m_1, called the initial parameter; m_2, called the terminal parameter; and m_3, called the incrementation parameter, are each either an integer constant or integer variable reference. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of 1 is implied for the incrementation parameter. At time of execution of the DO statement, m_1, m_2, and m_3 must be greater than zero.

The DO statement is a command to repeatedly execute the statements that follow, up to and including statement number n. The first time the statements are executed, i is equal to m_1. Each succeeding time the statements are executed, i is increased by m_3. When i is equal to the highest value not exceeding m_2, control passes to the statement following statement number n.

The range of a DO is defined by the set of statements that will be executed repeatedly; it is the sequence of consecutive statements immediately following the DO, up to and including the statement numbered n.

The index of a DO is the integer variable i, which is controlled by the DO in such a way that its value begins at m_1 and is increased each time by m_3 until further incrementation would cause the value of m_2 to be exceeded. Throughout the range it is available for computation, either as an ordinary variable or as the variable of a subscript. After the last specified execution of the range, the DO is said to be satisfied.



```
          :
   10 DØ 11 I=1,10
   11 A(I) = I * N(I)
   12     :
          :
```

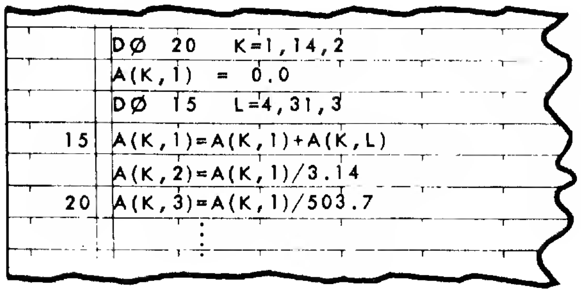
Assume, for example, that control has reached statement 10 of the following partial program.

The range of the DO is statement 11, and the index is I. The DO sets I to 1 and control passes into the range. The value of 1*N(1) is computed, converted to a real number, and stored in A (1). Because statement 11 is the last statement in the range of the DO and the DO is unsatisfied, I is increased to two and control returns to the beginning of the range (also statement 11), where 2*N(2) is computed and stored in A(2). This iteration process continues until statement 11 has been executed with I equal to 10. Control is then passed to statement 12 since the DO loop has been satisfied.

Nested DO's

Among the statements in the range of a DO loop may be other DO statements. When such is the case, all of the statements in the range of the latter DO loop must also be in the range of the former. A group of DO loops satisfying this rule is called a nest of DO's. Figure 3–1 illustrates this rule.

Example of Nested DO's



```
      DØ 20   K=1,14,2
      A(K,1) = 0.0
      DØ 15   L=4,31,3
   15 A(K,1)=A(K,1)+A(K,L)
      A(K,2)=A(K,1)/3.14
   20 A(K,3)=A(K,1)/503.7
        :
```

It is possible for a terminal statement to be the terminal statement for more than one DO statement. The following example shows this type of nested DO statements used to sum a triangular array. The value summed is

X(1,1) + X(2,1) + X (2,2) + X(3,1) +X(3,2) +X(3,3) +X(4,1) +X(4,2) +X(4,3) +X (4,4) +. . .+X(10,9) +X(10,10)

```
      SUM = 0.0
      DØ 40 I=1,10
      DØ 40 J=1,I
   40 SUM = SUM + X (I,J)
      :
```

As mentioned previously, the terminal statement cannot be a STOP, PAUSE, DO, RETURN, IF or GO TO statement or logical IF containing one of these forms because execution of these statements would cause the DO-loop to lose control.

CONTINUE

The CONTINUE statement causes no action and generates no coding. It is used for terminating DO loops and its form consists of the single word

CONTINUE.

```
      DØ 32 M=2,23
      IF(X(M)-10.0)30,32,31
   30 X(M) = SIN(X(M))+3.14
      GØ TØ 32
   31 X(M) = 3.14 - CØS(X(M))
   32 CØNTINUE
      :
```

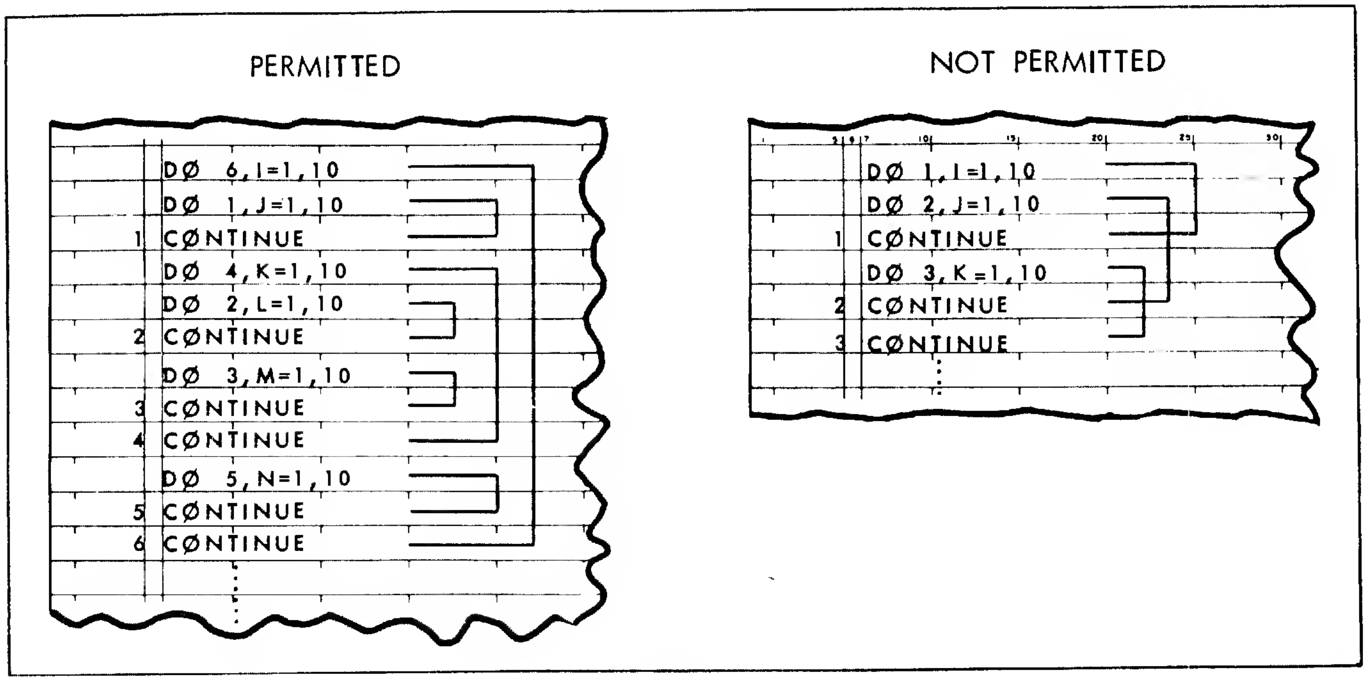
PAUSE

This statement is useful when the run-time program must stop temporarily for operator action such as changing data tapes. A PAUSE statement takes one of the following forms:

PAUSE m
or
PAUSE

where m is an identification constant and is an integer value in the range (0 ≤ m < 32768).

The PAUSE statement transfers to the F$HT subroutine which types the message PAUS (the DDP-116, -516 types the letters PA) and then halts the program with the value m in the accumulator. The identification constant m, when included, usually indicates the particular PAUSE statement that caused the halt. Pressing the computer's start button causes the program to continue, starting with the first statement following the PAUSE statement.



PERMITTED

```
      DØ 6,I=1,10
      DØ 1,J=1,10
    1 CØNTINUE
      DØ 4,K=1,10
      DØ 2,L=1,10
    2 CØNTINUE
      DØ 3,M=1,10
    3 CØNTINUE
    4 CØNTINUE
      DØ 5,N=1,10
    5 CØNTINUE
    6 CØNTINUE
      :
```

NOT PERMITTED

```
      DØ 1,I=1,10
      DØ 2,J=1,10
    1 CØNTINUE
      DØ 3,K=1,10
    2 CØNTINUE
    3 CØNTINUE
      :
```

FIGURE 3—1

STOP

The STOP statement is placed at the logical end of the program and causes the computer to halt.

The STOP statement uses one of the following forms:

STOP m
or
STOP

where m is an identification constant and is an integer value in the range ($0 \leq m < 32768$).

This statement transfers to the F$HT subroutine that types the message STOP (the DDP-116 types the letters ST) and then halts the program with the value m in the accumulator. The identification constant m, when included, usually indicates the particular STOP statement that caused the halt. Pressing the computer's start button will cause the message to be retyped. There is no return from the STOP statement.

END

The statement placed last physically in every program must be the END statement that consists of the word

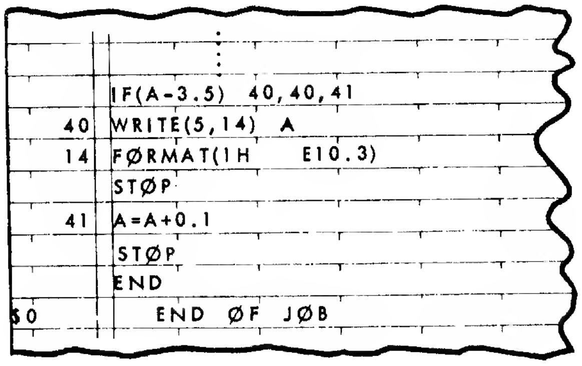
END.

It signals the compiler that the program is complete and no additional statements remain to be processed. No coding is generated by this statement, although at this time the compiler assigns all variables and constants needed by the program compiled. There must be END statement in every program.

END OF JOB

If a program job consists of just one program, the END statement is followed by an END OF JOB card that consists of dollar sign in column 1 and the digit 0 in column 2. The rest of the line is ignored and may be used for comments.

If a programming job consists of a FORTRAN program and several FORTRAN subprograms, they may be grouped together with the program first followed by the subprograms. The program and each subprogram is terminated by an END statement; and the last subprogram in the group is also followed by an END OF JOB card (line).



```
              :
              :
       IF(A-3.5)  40,40,41
   40  WRITE(5,14)  A
   14  FØRMAT(1H    E10.3)
       STØP
   41  A=A+0.1
       STØP
       END
$0         END  ØF  JØB
```

SECTION IV
SPECIFICATION STATEMENTS

Specification statements are considered non-executable statements in that they do not generate any object program instructions, but organize or classify data to be used by subsequent statements. All specification statements must appear before any statement that generates coding (executable statements). There is no "ordering" needed within a series of specification statements except for the DATA statement which must appear last.

SPECIFICATION STATEMENTS	DESCRIPTION
INTEGER	Declares mode of variables is INTEGER.
REAL	Declares mode of variables is SINGLE PRECISION Floating Point (Real).
DOUBLE PRECISION	Declares mode of variables is DOUBLE PRECISION Floating Point.
COMPLEX	Declares mode of variables is COMPLEX FLOATING POINT.
LOGICAL	Declares mode of variables is logical.
DIMENSION	Declares arrays and sets their sizes.
EXTERNAL	Declares subroutine names that are to be used as arguments for other subroutines.
EQUIVALENCE	Shares storage assignments of variables and/or arrays.
COMMON	Assigns variable and/or array storage to a common area.
DATA	Sets variables and/or array elements to initial values.

INTEGER STATEMENT

Unless otherwise declared, a variable, array, or function whose first letter is I, J, K, L, M or N is of the INTEGER mode. Other variables, arrays, or functions may also be mode-classified as INTEGER, regardless of first letter, by including them in the list of an INTEGER statement.

```
      INTEGER   XI,ALPHA,ARRAY,SUBR,XABS
```

REAL STATEMENT

Unless otherwise declared, a variable, array, or function whose first letter is not I, J, K, L, M or N is of the SINGLE PRECISION floating point (REAL) mode. Other variables, arrays, or functions may also be mode-classified as REAL, regardless of first letter, by including them in the list of a REAL statement.

```
      REAL   I3,NUMBER,LIST,LØG,INT
```

DOUBLE PRECISION STATEMENT

All variables, arrays, or functions that are to be of the DOUBLE PRECISION floating point mode, must be included in the list of a DOUBLE PRECISION statement.

```
      DØUBLE PRECISIØN   A4,TABLE,DSIN,DSQRT
      DØUBLE PRECISIØN   X,INCR,MØNEY
```

COMPLEX STATEMENT

All variables, arrays, or functions that are to be of the COMPLEX floating point (or COMPLEX) mode, must be included in the list of a COMPLEX statement.

```
      CØMPLEX   TEST,DECK,CSQRT,CABS,X
```

LOGICAL STATEMENT

All variables, arrays, or functions that are to be of the LOGICAL mode, must be included in the list of a LOGICAL statement.

```
      LØGICAL   L1,L2,BØØL,PILE,BETA,ETC
      DIMENSIØN   LIST(400),TABLE(10,10,4),ARRAY(40,10)
```

DIMENSION STATEMENT

The DIMENSION statement is used to declare arrays and to define their sizes. A DIMENSION statement takes the form:

DIMENSION $v_1(i_1)$, $v_2(i_2)$, ... $v_n(i_n)$

Each v is an array declarator where v is the name of the array. The mode of each item in the array is determined by the first letter of the name of the array or by including the name in one of the declaration statements (INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL). The subscript "i" is comprised of one, two, or three positive non-zero integer constants, or dummy variables separated by commas. The number of constants represents the number of dimensions by which the array is referenced; and the value of each constant represents the maximum size of each dimension.

When arrays are passed to subprograms, the subprogram must re-declare the array. The mode, number of dimensions, and size of each dimension must agree with that declared by the calling program, but the name of the array need not agree.

```
      DIMENSIØN   LIST(400)
         :
      CALL CHKSUM (LIST,ISUM)
         :
```

```
      SUBRØUTINE  CHKSUM(ITEMS,K)
      DIMENSIØN   ITEMS(400)
         :
```

It is possible for one or more of the subscripts in a DIMENSION statement to be an integer variable instead of a constant. This situation is only possible in a FORTRAN subfunction where the calling program provided (and declared) the array name and all variable subscripts. In other words, for any variable-dimensioned array given in a subfunction's DIMENSION statement, both the array's name and all variable subscripts must be dummy names.

This feature is useful for general-purpose subprograms that manipulate arrays (such as a matrix multiply subroutine). Since the subprogram could work with any size array (given enough memory), it lets the calling program specify the size of array, rather than specifying an array of constant size. The calling program, of course, must declare the array using constant subscripts.

```
      DIMENSIØN    ARRAY(40,40)
         :
      CALL MATMPY(ARRAY,40,X)
         :
```

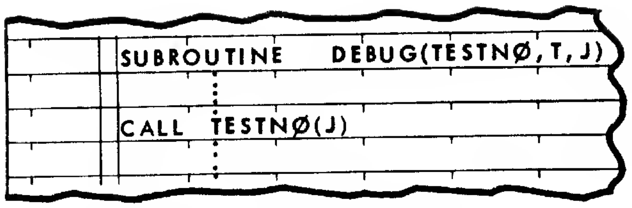
```
      SUBRØUTINE  MATMPY(TABLE,MAX,A)
      DIMENSIØN    TABLE(MAX,MAX)
         :
```

Other methods in which arrays may be declared and sized are discussed in the paragraph entitled "Alternate Methods of Declaring Arrays" later in this section.

EXTERNAL STATEMENT

One of the allowable types of arguments passed on to a subprogram is another subprogram's name. In order to do this, the subprogram name being used as an argument must be declared as a subprogram by placing the name in the list of an EXTERNAL statement. Only subprogram names used as arguments need to be declared by an EXTERNAL statement.

```
      EXTERNAL    TEST1,TEST2,TEST3
         :
      CALL DEBUG(TEST1,A,I)
         :
      CALL DEBUG(TEST3,X,I)
         :
      CALL DEBUG(TEST2,A,J)
         :
```



```
      SUBROUTINE   DEBUG(TESTNØ,T,J)
                   :
      CALL  TESTNØ(J)
                   :
```

EQUIVALENCE STATEMENT

The EQUIVALENCE statement is used to permit the sharing of memory storage by two or more entities. The general format of the EQUIVALENCE statement is:

EQUIVALENCE (k_1), (k_2), ... , (k_n)

where each k represents a list of two or more variable or subscripted variable or array names separated by commas. Each element in the list represented by k is assigned the same memory storage by the compiler. All subscripts appearing in an equivalence list must be integer, positive, nonzero constants.

An EQUIVALENCE statement may equate single variables to each other, entire arrays to each other, elements of an array to single variables, or vice versa as shown in Figure 4–1. An element of an array may be expressed in an EQUIVALENCE statement in one of two ways.

1. It may be expressed exactly as in a DIMENSION statement. Assume the element A (4,1) of the two-dimensional array A (4,2) is to be equated to variable B (6) of the one-dimensional array B (10), the statement could be:

EQUIVALENCE (A (4,1), B (6))

2. It may be expressed as the equivalent fictitious single-dimensional subscript that indicates the order in which the element is stored in memory. Again assuming element A (4,1) is to be equated to variable B (6), the statement could be written:

EQUIVALENCE (A (4), B (6))

where A (4) specifies that the element A (4,1) is stored in the fourth location of the storage block reserved for the two-dimensional array A (4,2).

The mode assigned to each element determines the number of memory cells occupied by each element as shown in Table 4–1.

If an INTEGER or LOGICAL mode variable is made equivalent to a REAL, DOUBLE PRECISION or COMPLEX mode variable, the former variable shares memory storage with the first of the words required by the later variable. Figure 4–1 shows the relative memory assignments caused by an EQUIVALENCE statement.

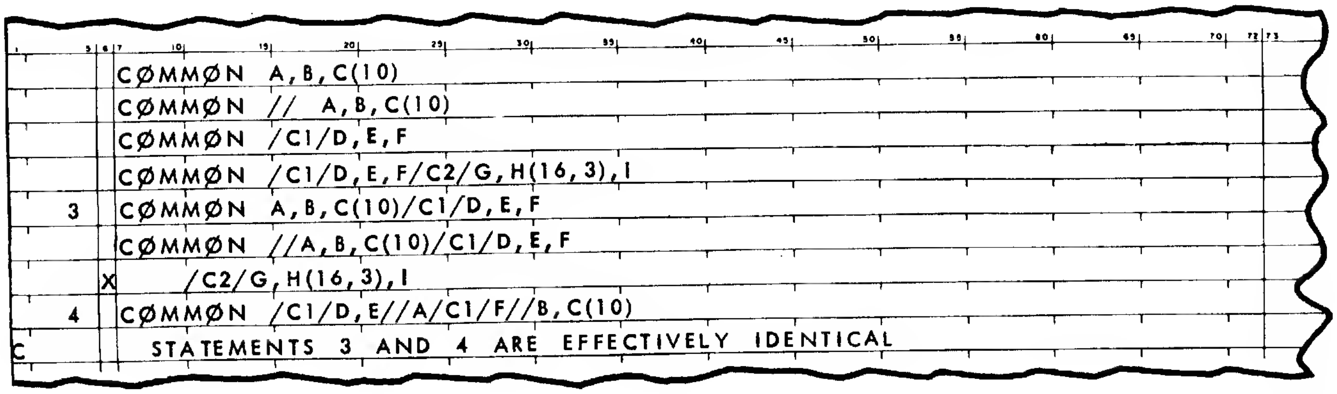
Variables and array elements appearing in EQUIVALENCE statements may also appear in COMMON statements. The resulting effect is explained in the following paragraph. Dummy arguments for a subprogram cannot be used as elements within an EQUIVALENCE statement contained in that subprogram.

TABLE 4–1.
MODE/MEMORY CELLS CORRESPONDENCE

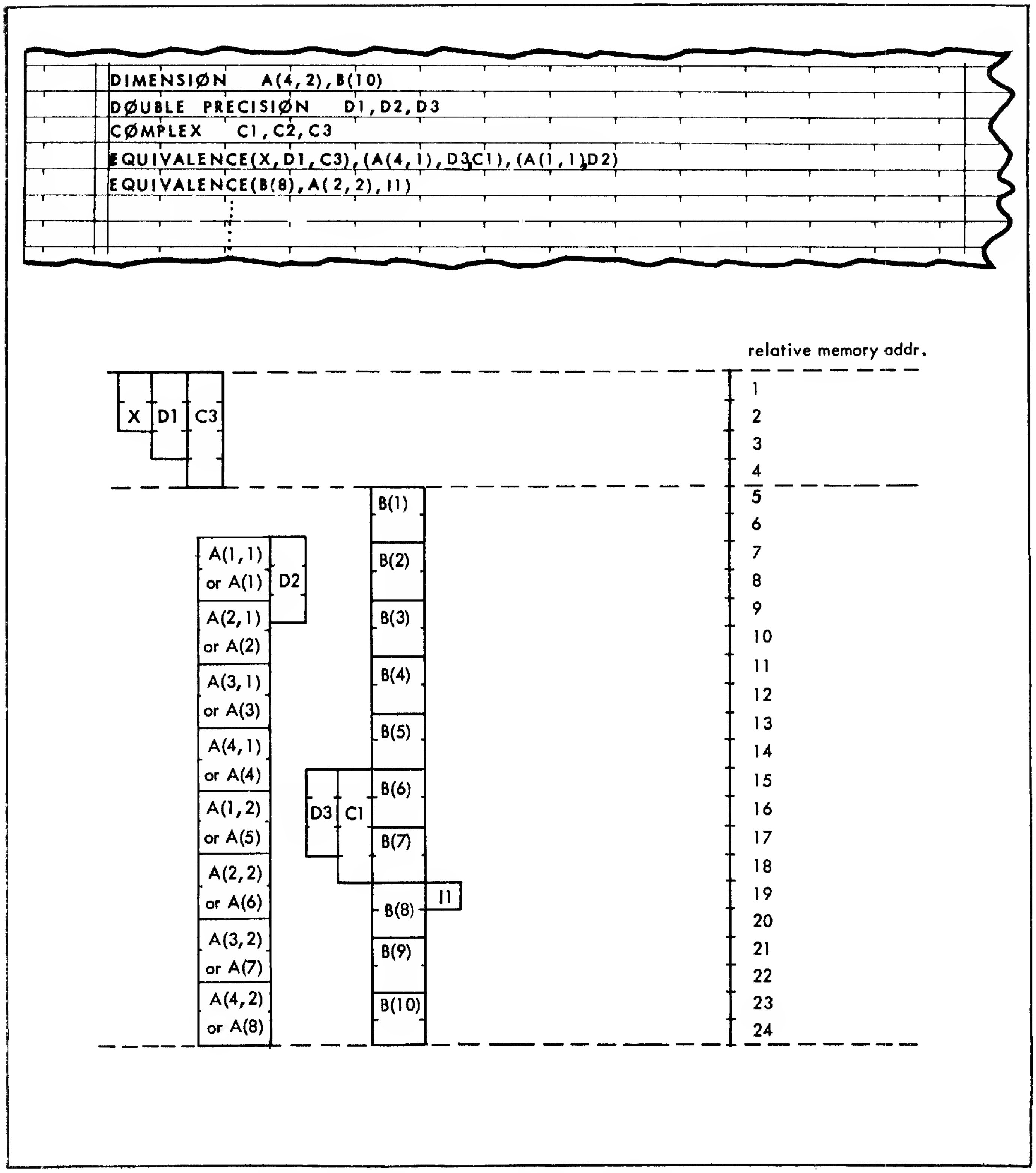
MODE ASSIGNMENT	NUMBER OF MEMORY WORDS
INTEGER	1
REAL	2
DOUBLE PRECISION	3
COMPLEX	4
LOGICAL	1

COMMON STATEMENT

The COMMON statement provides a means of sharing memory storage among subprograms or transferring data between subprograms or between segments of a chained program. The format of the COMMON statement is:



```
      CØMMØN  A,B,C(10)
      CØMMØN  //  A,B,C(10)
      CØMMØN  /C1/D,E,F
      CØMMØN  /C1/D,E,F/C2/G,H(16,3),I
    3 CØMMØN  A,B,C(10)/C1/D,E,F
      CØMMØN  //A,B,C(10)/C1/D,E,F
     X    /C2/G,H(16,3),I
    4 CØMMØN  /C1/D,E//A/C1/F//B,C(10)
C      STATEMENTS 3 AND 4 ARE EFFECTIVELY IDENTICAL
```



```
DIMENSIØN    A(4,2),B(10)
DØUBLE PRECISIØN   D1,D2,D3
CØMPLEX   C1,C2,C3
EQUIVALENCE(X,D1,C3),(A(4,1),D3,C1),(A(1,1),D2)
EQUIVALENCE(B(8),A(2,2),I1)
```

FIGURE 4—1. MEMORY ASSIGNMENTS CAUSED BY EQUIVALENCE STATEMENT

COMMON $a_1, a_2, \ldots, a_n$

COMMON/x_1/$a_1, a_2, \ldots, a_n$/x_n/$a_1, a_2, \ldots, a_n$

where a_i specifies a list of variable names or array names and x is a COMMON block name. If a specifies an array name, it may be followed by its dimensioning information in parentheses. COMMON block names, when specified, are enclosed between two slashes. A COMMON block name may be unspecified (called blank common) and if such a block appears first in a COMMON statement, the two slashes may be omitted. Names of COMMON blocks must not be identical with the name of a subprogram called on by the program job, or the name of a subroutine on the library tape. The following example illustrates some acceptable COMMON statements.

The data items within a COMMON block are assigned sequentially in the order of appearance. The actual location of a COMMON block is made by the loader program in such a way that all COMMON blocks with the same name are assigned to the same area regardless of the program or subprogram in which they are defined. The loader program is designed to assign all blank common data in such a way that it overlaps the loader program, thereby making the memory area occupied by the loader program available for data storage.

Elements within a COMMON block may be overlapped and interrelated by listing such elements in a group. If an element included within an EQUIVALENCE statement group is also specified to be in a COMMON area, all other items within the equivalence group are also in that COMMON area.

The number of words that a COMMON block occupies depends on the number of elements, the mode of the elements, and the interrelations between the elements specified by an EQUIVALENCE statement. COMMON blocks that appear with the same block name (or no name) in various programs or subprograms of the same job are not required to have the elements within the block agree in name, mode, or order; but the blocks *must* agree in total words occupied.

Figures 4–2 and 4–3 show the relative memory assignments caused by COMMON statements.

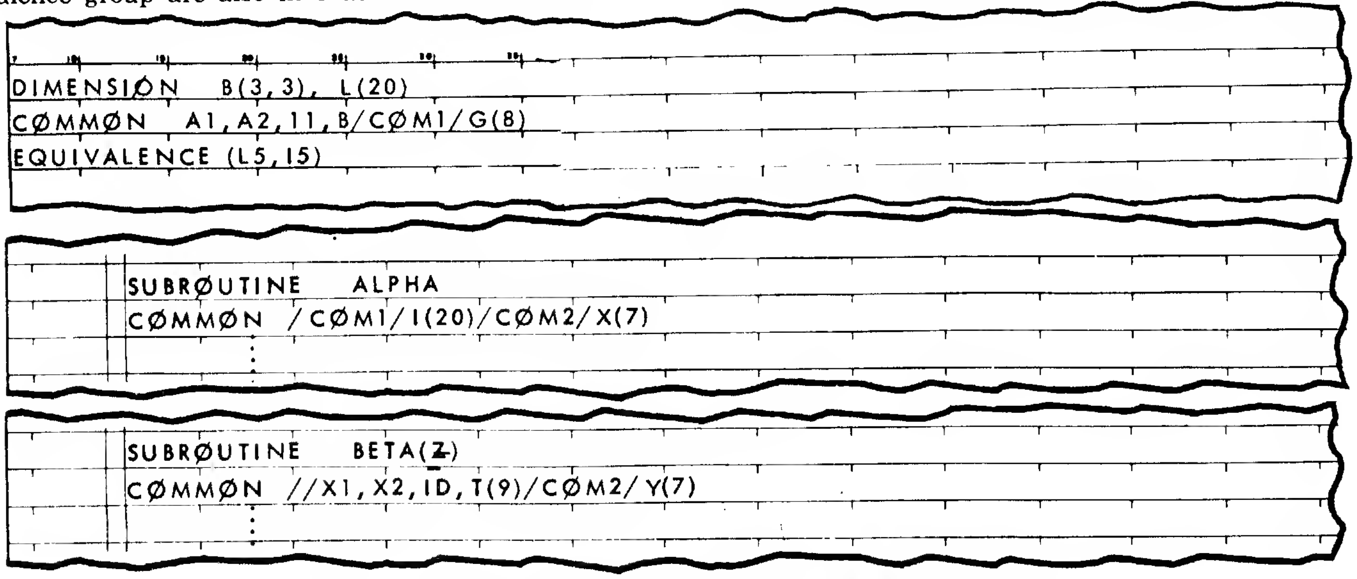
DATA STATEMENT

The DATA statement is used to set variables or array elements to initial constant values during loading of the object program. (The variables are not re-initialized if the program is restarted without reloading.) A DATA initialization statement is of the form:

DATA $k_1/d_1/, k_2/d_2/, \ldots\ldots, k_n/d_n/$

Each k is a list containing non-dummy names of variables or array elements (with constant subscripts) separated by commas. Each d is a corresponding list of constants with optional signs.

There must be a correspondence in order and mode between the name list and the data list. If the data list consists of a sequence of identical constants, the constant need only be written once and preceded by the number of repeats (integer constant)



```
DIMENSIØN  B(3,3), L(20)
CØMMØN  A1,A2,I1,B/CØM1/G(8)
EQUIVALENCE (L5,I5)
```

```
      SUBRØUTINE   ALPHA
      CØMMØN /CØM1/I(20)/CØM2/X(7)
         :
```

```
      SUBRØUTINE   BETA(Z)
      CØMMØN //X1,X2,ID,T(9)/CØM2/Y(7)
         :
```

FIGURE 4—2. COMMON STATEMENTS EFFECTING MEMORY ASSIGNMENTS IN FIGURE 4—3

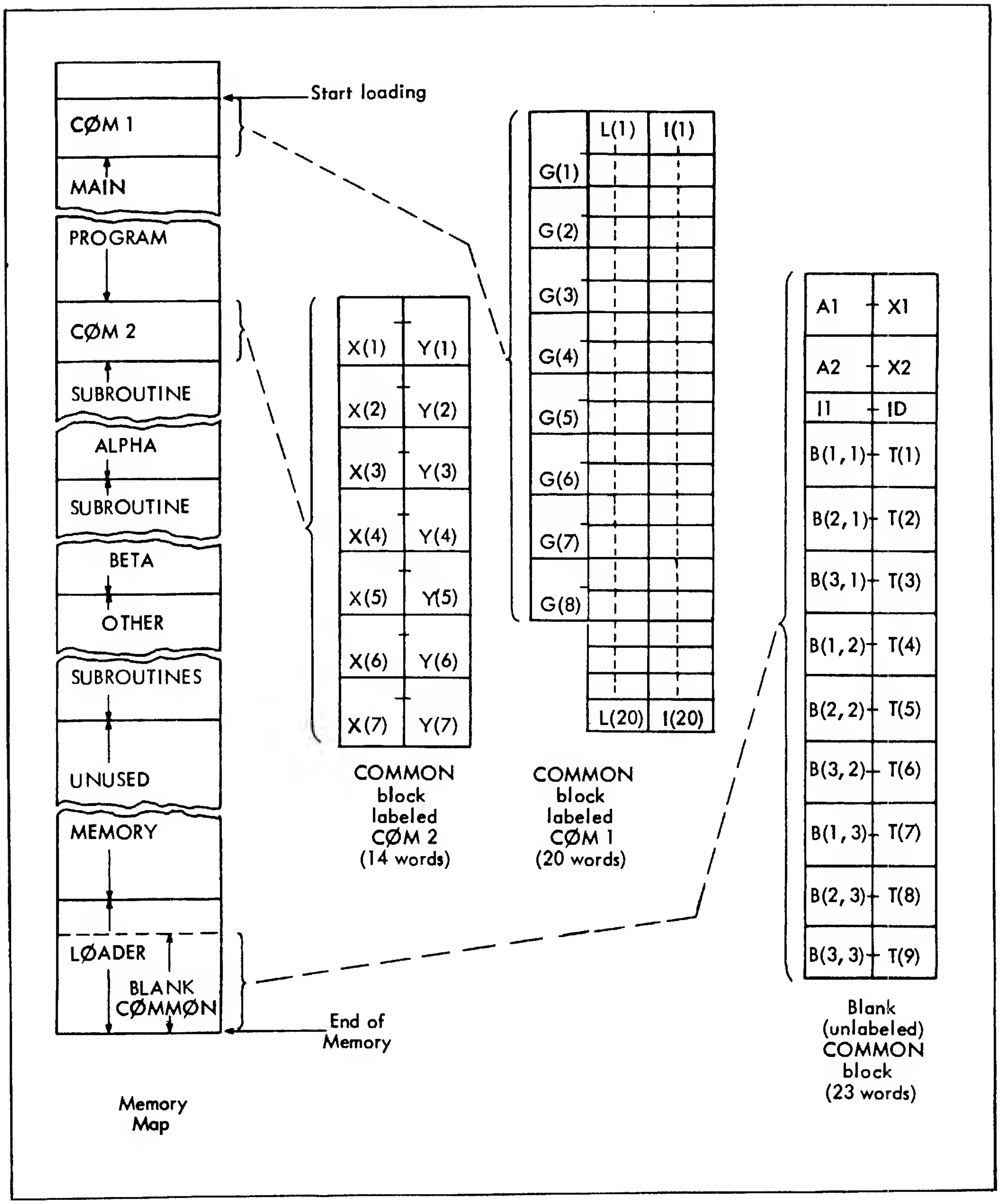


FIGURE 4—3. MEMORY ASSIGNMENTS CAUSED BY COMMON STATEMENTS IN FIGURE 4—2

INTEGER	/13,-4179,1,+6,6*0/
REAL	/13.0,-4.179E3,100E-2,+6.0,6*0./
DOUBLE	/13.D,-4.179D3,100D-2,+0.6D1,6*0.D/
COMPLEX	/(13.0,-4.179E3),(100E-2,+6.0),6*(0.,0.)/
LOGICAL	/.TRUE.,.TRUE.,.FALSE,
HOLLERITH	/4HDEC.,3HDEG
MIXED	/13,13.0,1.3E2,13.D,(13.0,0.0)/

FIGURE 4—4

and an asterisk character. For example:

/1.4,3*2.0,0.0/=/1.4,2.0,2.0,2.0,0.0/

A Hollerith constant may appear in the data list as a string of characters preceded by the letter H that, in turn, is preceded by a constant indicating the string length. The characters will be stored in their BCD code, left justified if necessary. Types of constants allowed within the data list are shown in Figure 4–4.

Consider the first DATA statement shown in the example below. This statement assigns the value 0.10762 to A1 (4), 1.0E5 to X, 1 to 1, etc. The assignment is done at load time not at execution time. A DATA statement is not executable.

Restrictions

The DATA statement(s) must appear as the last specification statement(s). Variables or array elements assigned to unlabeled (blank) COMMON may not be initialized by the DATA statement. Variables or array elements assigned to labeled COMMON areas may be initialized by a DATA statement but only within a BLOCK DATA subprogram (see Section VI).

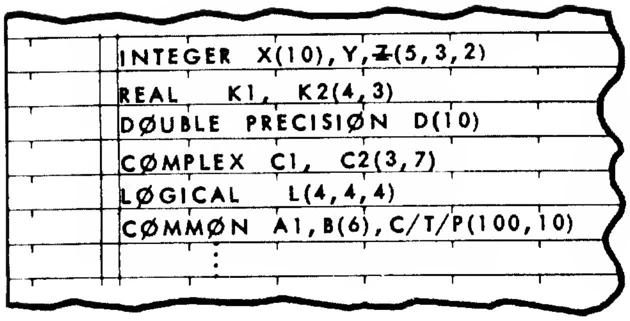
```
      DIMENSIØN   A1(10),  A2(10)
      DØUBLE  PRECISIØN   D1
      LØGICAL   L1,L2,L3
      CØMPLEX   C1
      DATA   A1(4),X,I,D1,L1,L2,
     X      C1/0.10762,1.0E5,1,1.0D,
     X      .TRUE.,.TRUE.,(4.3,0.0)/
      DATA   P/14.9/,  Q/0.1E-3/,
     X      R/0./,  J/4095/,  L3/.F./,
     X      A2(1),A2(2),A2(3),A2(4),
     X    A2(5),A2(6),A2(7),A2(8),
     X    A2(9),A2(10)/10*0.0/
```

ALTERNATE METHODS OF DECLARING ARRAYS

The size and mode of an array is normally declared by a DIMENSION statement. However, it is sometimes more convenient to declare arrays by other specification statements. This is allowed in the INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and COMMON statements. The specification of the size of the array is made in the same manner as in the DIMENSION statement, by following the array name with the maximum size of each dimension within parenthesis.

The following two examples accomplish the same task of declaring arrays and typing variables. The second example, however, does not use a DIMENSION statement. Both methods are acceptable.

```
      INTEGER   X,Y,Ƶ
      REAL   K1,K2
      DØUBLE  PRECISIØN  D
      CØMPLEX  C1,C2
      LØGICAL  L
      CØMMØN   A1,B,C/T/P
      DIMENSIØN  X(10),K2(4,3),
     1   Ƶ(5,3,2),D(10),C2(3,7),
     2   L(4,4,4),B(6),P(100,10)
           :
```



```
INTEGER X(10),Y,Z(5,3,2)
REAL   K1, K2(4,3)
DØUBLE PRECISIØN D(10)
CØMPLEX C1, C2(3,7)
LØGICAL   L(4,4,4)
CØMMØN A1,B(6),C/T/P(100,10)
        :
```

TRACE STATEMENT

The TRACE statement causes coding to be inserted into the object program, when specified variables or array names are defined by an arithmetic statement, or after every variable definition within a specified area. The effect of the inserted coding and examples of TRACE statement formats are described in Section VII. The TRACE statement has two acceptable formats: item trace and area trace.

Item Trace

Item trace takes the form:

TRACE $x_1, x_2, \ldots \ldots x_m$

where each x represents a variable or array name. Special coding is inserted after each time one of the list names is defined by an arithmetic statement.

Area Trace

Area trace takes the form:

TRACE n

where n represents a single statement number. Trace coding is inserted after all statements starting with this statement, up to and including statement number n.

SECTION V
INPUT/OUTPUT STATEMENTS AND FORMAT SPECIFICATIONS

INPUT/OUTPUT STATEMENTS

There are two main types of input/output statements:

(1) READ and WRITE statements

(2) I/O Control statements

This section deals primarily with the former, and with associated format specifications.

In the statement descriptions in Table 5—1, the arguments "u", "f", and "list" are used as follows:

The argument *u* represents the input/output unit number and may be an integer constant between 0 and 9, or an integer variable. Standard assignment of device numbers is as follows:

0 =	spare	5 =	magnetic tape no. 1
1 =	typewriter	6 =	magnetic tape no. 2
2 =	paper tape	7 =	magnetic tape no. 3
3 =	cards	8 =	magnetic tape no. 4
4 =	line printer	9 =	magnetic tape no. 5

(Changing device assignments is discussed in Appendix B.)

The argument *f* represents the format statement reference and must be a format statement number if the format statement is included in the program being compiled, or an array name if the format

TABLE 5—1.
I/O STATEMENTS

TYPE	STATEMENT	DESCRIPTION
Formatted READ	READ (u,f) list, or READ (u,f)	Input from unit u according to format statement f, and assign the resulting values to the list elements (if any).
Formatted WRITE	WRITE (u,f) list, or WRITE (u,f)	Convert the list elements according to format statement f, and output using device u.
Unformatted READ	READ (u) list, or READ (u)	Input the next record from device u in binary format and, if there is a list, assign these values to corresponding items on the list. If the record contains more items than are required for the list, the rest of the record is lost. If more information is required for the list than is in one record, additional records will be read. If no list is present, one record will be read but ignored (resulting in an effective forward skip action).
Unformatted WRITE	Write (u) list	Write all words specified by the list in binary format without converting. Writing is performed on device u. If the list elements do not fill one record, the rest of the record is padded with zero bits. If the list elements require more than one record, multiple records will be written and the last record padded with zero bits if necessary.
Control Statements	REWIND u	Rewind magnetic tape unit u to its initial start point. Only unit 5, 6, 7, 8 or 9 (magnetic tape 1, 2, 3, 4 or 5) should be specified.
	BACKSPACE u	Backspace magnetic tape unit u by one record (unless it is at its initial start point). Only unit 5, 6, 7, 8 or 9 (magnetic tape 1, 2, 3, 4 or 5) should be specified.
	ENDFILE u	Punch a stop code on paper tape if the unit number is 2, or write an ENDFILE record on magnetic tape 1, 2, 3, 4 or 5 if the unit number is 5, 6, 7, 8 or 9. When an ENDFILE record or stop code character is read during input, a coded message will be typed followed by a halt.

TABLE 5—2.
EXTERNAL RECORDS

UNIT (DEVICE)	FORMATTED (BCD) RECORD DEFINITION	UNFORMATTED (BINARY) RECORD DEFINITION
Typewriter	One line of type terminated by a carriage return. Maximum of 120 characters/line (72 characters/line on DDP-116, -516)	Undefined
Line Printer	One line of printing with a maximum of 120 characters/line	Undefined
Cards	One card, 80 characters	One card, 40 words (60 words on DDP-116, -516)
Paper Tape	One card image of 80 characters	One card image of 40 words (60 words on DDP-116, -516)
Magnetic Tape	Line printer image 120 characters	One card image of 40 words (60 words on DDP-116, -516)

statement is to be entered into the array at execution time.

The argument *list* represents the list of individual arguments that are to be input or output.

EXTERNAL RECORDS

The information transmitted between computer storage and external media by one input/output statement is separated into physical groups called records. The definition of a record varies with the I/O device, the type of machine, and whether the record is BCD or binary. Table 5–2 defines the data that comprises a record for each I/O unit. It is the programmer's responsibility to avoid exceeding the maximum record size for a given device.

External formatted records may contain the values of several variables; each variable is considered a field of information. In order to specify the manner of conversion on input or output, it is necessary to state the size or width of the data field. If the variable is floating point, it is also necessary to state the position of the decimal point. The width of the field is the total number of characters necessary to describe the information on a coding form. However, fields may be larger if leading blanks are included, or smaller if truncation is desired.

If formatted input records exceed the maximum number of characters allowed for a particular input device, all characters after the maximum are undefined. The next input record is not automatically read. A means of processing multiple records is described later in this section.

I/O STATEMENT ARGUMENT LISTS

Simple Arguments

Most input/output statements include a list of the quantities to be transmitted. The list is ordered, and its order must be the same as the order in which the words of information exist (for input), or will exist (for output), in the external medium.

The items within the list may be variables, subscripted variables, or array names. Constants are not allowed as list items.

When an integer variable is used as a subscript to a variable in the list of a READ statement, it must be previously mentioned as a variable in that same list (physically to the left of where it appears as a subscript). Following is an example of two acceptable lists.

```
      DIMENSIØN   X(100),Y(10,10)
      READ(1,20)  A,  X(3),I,Y(I+1,4)

      DIMENSIØN   X(100),  Y(10,10)
      READ(1,20)  A,  X(3), I, Y(I+1,4), X, B
                   :
```

Implied DO-Loops

A group of simple arguments can be enclosed in parentheses and include control information which specifies the number of times the group is to be repeated. This control information is very similar in format to that used by the DO statement. For example:

```
      DIMENSIØN B(40),C(10),D(10,4),E(10,4),F(4,4)
      WRITE(5,20)  K,  B(3),(C(I),D(I,K),I=1,10),
     X        ((E(I,J),I=1,10,  ),F(J,3),J=1,K)
                   :
```

If the list shown in the preceding example is used with an output statement, the information is written in the external medium in the following order:

```
K,B(3),C(1),D(1,K)C(2),D(2,K),......,
                C(10),D(10,K),
  E(1,1),E(2,1),......,E(10,1),F(1,3),
  E(1,2),E(2,2),......,E(10,2),F(2,3),
                       ......,F(K,3).
```

Similarly, if the list is used with an input statement, the successive words, as they were read from the external medium, are placed in the sequence of storage locations specified above.

Thus, the list reads from left to right with the repetition of variables enclosed in parentheses. Only variables, not constants, may be listed. The repetition is identical to that of a DO loop, as if each open parenthesis (except subscripting parentheses) were a DO, with indexing given immediately before the matching closing parentheses, and with range extending up to that indexing information. The order for the above list is the same as for the "program":

```
1. K
2. B(3)
3. DO5I = 1,10
4. C(I)
5. D(I,K)
6. DO 9 J = 1,K
7. DO 8 I = 1,10
8. E(I,J)
9. F(J,3)
```

Notice that indexing information, as in DO's, consists of three constants or integer variables, and that the last of these may be omitted, in which case it is understood to be one.

For a list of the form K, (A(K)) or (A(I),I = 1,K), where an index or indexing parameter itself appears earlier in the list of an input statement, the indexing is carried out with the newly read-in value.

SPECIFICATION OF MATRIX ORDER

FORTRAN, in effect, considers variables in a matrix such that a list for either input or output in the form

((A(I,J),J = 1,3), I = 1,2)

specifies that I x J items of information are transmitted in the order

$A_{1,1},A_{1,2},A_{1,3},A_{2,1},A_{2,2},A_{2,3}.$

The above is the order in which the items are output; for input, it is the order in which the data should be written on the data sheet. If it is desired to write the data by columns or to print the items by columns, the list takes the form

((A(I,J),I = 1,2), J = 1,3)

It can be seen here that it is the inner-most index, rather than the inner-most subscripted variable, that determines which subscript varies more rapidly.

INPUT/OUTPUT OF ENTIRE ARRAYS

When input/output of an entire matrix is desired, an abbreviated notation may be used for the list of the input/output statements. Only the name of the array need be given and the indexing information may be omitted. Thus the list

A

is sufficient to cause the read-in of all the items of matrix A in their natural order. The natural order is considered to be

((A(I,J),I = 1,2)J = 1,3)

In such a case, FORTRAN examines to see whether or not the variable has been defined as an array. If such a definition is not made, only a single element will be transmitted.

FORMAT STATEMENT

The form of the FORMAT statement is

FORMAT (S_1, S_2, ..., S_n)

where S_i are descriptions of the external form of the variables comprising a record. The S_i provide the compiler with the information necessary for conversion from and to external form.

A FORMAT specification describes the record to be converted by giving, for each field in the record (from left to right, beginning with the first character), a basic field specification written in the form:

nKw.d

where:

1. The letter n represents a positive integer indicating the number of successive fields within one unit record which are to be converted according to the same specification. If n is equal to 1, it may be omitted.

2. The letter K represents a control character specifying the type of conversion to be used. This character may be I,E,F,G,D,P,L,A,H or X.

3. The letter w represents the width of the field. The field widths may be made greater than neces-

sary to provide spacing blanks between the items on a line. Thus, a field specification of nK12, where only four digits are to be printed, would result in eight blanks preceding the digits. Within each field the printed output will always appear in the rightmost positions.

4. The letter d represents the number of positions in the field which appear to the right of the decimal point; it is used only with E, F, G and D-type conversions.

Within the unit record, field specifications are separated by commas.

Iw, Ew.d, nX, Fw.d, nAw, etc.

EXCEPTION: A comma need not follow a field specified by an H or X control character.

SUMMARY OF CONTROL CHARACTERS

There are nine different types of control characters, seven of which provide for the conversion of data between the internal machine language and the external notation as follows:

INTERNAL	TYPE	EXTERNAL
Integer variable	I	Decimal integer
Real variable	E	Floating-point, scaled
Real variable	F	Floating-point, mixed
Real variable	G	Floating-point, mixed/scaled
Double precision variable	D	Floating-point, scaled
Logical variable	L	Letter T or F
Real variable	A	BCD characters

Complex variables are represented by types E, F or G and appear externally as two floating-point numbers.

An eighth control character, X, provides for skipping characters in input or the specification of blank characters in output.

A ninth control character, H, designates Hollerith or heading fields. It may be used to output alphanumerical characters originating in the source program and for carriage control in printing or typing.

I-Type Conversion (Field Specification Iw or nIw)

The number of characters specified by w is converted as a decimal integer.

I-Output. Negative numbers have a minus sign before their first significant digit; no sign indicates a positive number. The digits are right justified with leading blanks. If an integer is truncated because w is not large enough, the sign position contains a dollar sign ($) if positive or an equals sign (=) if negative to indicate that some digits are missing.

In the following examples, the letter "b" indicates a blank (or space):

OUTPUT FORMAT	INTERNAL NUMBER	CHARACTERS OUTPUT
I8	+12345	bbb12345
I7	−12345	b−12345
I6	+12345	b12345
I5	−12345	=1234
I4	+12345	$123
I3	−12345	=12
I2	+12345	$1
I1	−12345	=
I0	+12345	(NO OUTPUT)
I5	+00000	bbbb0

I-Input. An input field of w characters is converted to a 23-bit plus sign binary integer (15-bit plus sign on the DDP-116, -516). If no minus sign is present, the value is considered positive. No decimal point character is allowed. If blanks (or spaces) are present in the field, they must precede the first digit or sign. Integer values with a magnitude up to 8388607 are accepted (32767 maximum on the DDP-116, -516).

FORMAT DESCRIPTOR	CHARACTERS INPUT	INTERNAL NUMBER
I5	bbbbb	+00000
I5	bbbb1	+00001
I5	bbb+1	+00001
I5	−bb15	−00015
I5	bbb−3	−00003
I5	12345	+12345
I5	−1234	−01234

E-Type Conversion (Field Specification Ew.d or nEw.d)

The number of characters specified by w is converted as a floating-point number with the number of digits specified by d to the right of the decimal point.

E-Output. Output consists of a minus sign or blank (if positive), the digit 0, a decimal point, the most significant d digits of the number to be output, followed by the letter E and a two-digit-plus-sign decimal exponent. The leading sign position is replaced by a dollar sign ($) if positive or an equals sign (=) if negative to indicate that the output must be truncated because the w field specification is too small.

OUTPUT FORMAT	INTERNAL NUMBER	CHARACTERS OUTPUT
E14.4	+12.3456	bbbb0.1235Eb02
E13.4	−0.0012321	bb−0.1232E−02
E12.4	−0.176	b−0.1760Eb00
E11.4	+123456.	b0.1235Eb06
E10.4	−123456.	=0.1235Eb0
E9.4	+123456.	$0.1235Eb
E8.4	−123456.	=0.1235E
E7.4	+123456.	$0.1235
E6.4	−123456.	=0.123
E5.4	+123456.	$0.12
E4.4	−123456.	=0.1
E3.4	+123456.	$0
E2.4	−123456.	=0
E1.4	+123456.	$

E-Input. See the discussion of D-input in the paragraph entitled "D-Type Conversion."

F-TYPE CONVERSION (BASIC FIELD SPECIFICATION Fw.d or nFw.d)

The number of characters specified by w is converted as a floating-point mixed number, with the number of digits specified by d to the right of the decimal point.

F-Output. Output consists of a minus sign or blank (if positive), followed by the integer portion of the number, a decimal point, and d digits of the fractional portion of the number. The sign is replaced by a dollar sign ($) if positive or an equals character (=) if negative to indicate that w characters are not sufficient to hold the sign, the integer digits, the decimal point, and d fractional digits.

OUTPUT FORMAT	INTERNAL NUMBER	CHARACTERS OUTPUT
F6.3	+0.00123	b0.001
F6.3	+0.12468	b0.125
F6.3	−0.12468	−0.125
F6.3	+1.23456	b1.235
F6.3	−6.00000	−6.000
F6.3	+12.3456	$12.34
F6.3	−123.456	=123.4
F6.0	+123.456	bb123.
F6.0	+0	bbbb0.

F-Input. See the discussion of D-input in the paragraph entitled "D-Type Conversion."

G-TYPE CONVERSION (BASIC FIELD SPECIFICATION Gw.d or nGw.d)

The external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real number.

G-Output. The form of the output depends on the magnitude of the internal floating-point number. Comparison is made between the exponent (e) of the internal value and the number of significant digits specified (d) by the format descriptor. If e is greater than d, an E-type conversion is used. If e is less than or equal to d, an F-type conversion is used, but modified by the following formula: F(w-4). (d-e), 4X. (Four blanks, as specified by the 4X, are always appended to the value.) If the value to be represented is less than |.1|, the E-type conversion is always used.

The sign position is minus if negative or blank if positive. If w or d is such that the number could not be converted properly, the sign position contains a dollar sign ($) if positive or an equal sign (=) if negative to provide an indication.

Following are some correctly formatted output values which show how the conversion formulas are used to determine the output presentation.

OUTPUT FORMAT	INTERNAL NUMBER	CHARACTERS OUTPUT
G14.6	$.12345123 \times 10^0$	bb0.123451bbbb
G14.6	$.12345123 \times 10^4$	bbb1234.51bbbb
G14.6	$.12345123 \times 10^8$	bb0.123451E+08
G14.6	$.12345123 \times 10^{10}$	bb0.123451E+10

G-Input. See the discussion of D-Input in the following paragraph.

D-TYPE CONVERSION (BASIC FIELD SPECIFICATION Dw.d or nDw.d)

The external field occupies w character positions, the fractional part of which consists of d digits. The internal format is DOUBLE-PRECISION floating-point format (three words).

D-Output. The external output format is the same as E-output format except the letter E is replaced by the letter D.

OUTPUT FORMAT	INTERNAL NUMBER	CHARACTERS OUTPUT
D15.8	+12.34567890	b0.12345679Db02
D15.8	−0.0012321	−0.12321000D−02
D15.4	−9.176	bbbbb−.9176Db01
D11.4	+123456.	b0.1235Db06

D-Input. The external format of numbers input by the E, F, G or D input can be relatively loose. This format is identical for any of the inputs and is as follows:

1. Leading spaces (leading spaces are ignored)
2. A + or – sign (An unsigned input is assumed positive.)
3. A string of digits
4. A decimal point
5. A second string of digits (All non-leading spaces are considered to be zeros.)
6. The character D or E
7. A + or – sign
8. A decimal exponent

Each item in the list above is optional although it is obvious that if format 3 and 5 (above) are present, 4 is required; and that if format 8 is present, 6 or 7 (or both) is required.

Input data can be any number of digits in length, but must fall within the range of 0 to $\pm 10^{\pm 76}$ (0 to $\pm 10^{\pm 38}$ in the DDP-116, -516). Double-precision input retains 14 digits of significance (12 digits for the DDP-116, -516), and single precision input retains 7 digits of significance (same for the DDP-116, -516).

INPUT FORMAT	CHARACTERS INPUT	INTERNAL NUMBER
D12.4	bbbbbbbbbbbb	+0.0
D12.4	bbbbbbbbbbb3	+0.0003
D12.4	bbbbbbbbbb3.	+3.0
D12.4	1.0000000000	+1.0
D12.4	bbbbbb7bbbbb	+70.0
D12.4	bbbb2b3bb.bb	+20300.0
D12.4	bbbbb1.234E3	+1234.0
D12.4	bbbb1.234D3b	$+1.234*10^{30}$
D12.4	–12345678E–3	–1.2345678
D12.4	+123.40000–2	+1.234
E12.4	–0.1234567+4	–1234.0
E12.4	bbbbbb123456	+12.3456
F12.3	–bbbbb123456	–123.456
F12.2	bbbbbb123456	+1234.56
G12.1	bbbbb–123456	–12345.6
G12.0	bbbbb+123456	+123456.0

Note in the above examples, if no decimal point is given, an implied decimal point placed to the left of the first d places from the right is assumed. If a decimal point is included, the d specification is ignored.

Wherever the letter D appears in the examples in either the input format or characters input column, it may be replaced with the letter E, F, or G with the same result. All external numbers are converted to DOUBLE-PRECISION floating-point internal format, but are then truncated to SINGLE-PRECISION floating point format if required. In other words, a double-precision value can be input by either the E, F, G or D format descriptor, as can a single-precision value.

An error flag is set if any format errors, range errors, or unrecognized characters are input within the field processed during input. This flag may be checked and reset by the use of the OVERFL function. The result of such an input is undefined.

P Scale Factor (Basic Field Specification nP or –nP)

A scale factor is sometimes used preceding one of the control characters – D, E, F, or G – to effect a multiplication by some power of ten. When used, it is of the form

nPrKw.d or –nPrKw.d

where n determines the power of ten to be used, and r is the repetition number. Initially an implied scale factor of zero is assumed. When a P descriptor is processed, the scale factor specified (n) remains active for all format descriptors that follow until another P descriptor is processed. The value n must be present and is any unsigned integer (if positive), a minus sign followed by an integer (if negative), or zero. The n parameter must precede the letter P.

The effect of the current scale factor is different on input than on output.

Scale Factor Effect on Output

1. For E and D output, the fractional part is multiplied by 10^n and the exponent is reduced by n.

OUTPUT FORMAT	INTERNAL NUMBER	SCALE FACTOR	CHARACTERS OUTPUT
E12.4	+123456.	–5	bb0.0000Eb11
E12.4	–123456.	–4	b–0.0000Eb10
E12.4	+123456.	–3	bb0.0001Eb09
E12.4	–123456.	–2	b–0.0012Eb08
E12.4	+123456.	–1	bb0.0123Eb07
E12.4	–123456.	0	b–0.1235Eb06
E12.4	+123456.	1	bbb1.235Eb05
E12.4	–123456.	2	bb–12.35Eb04
E12.4	+123456.	3	bbb123.5Eb03
E12.4	–123456.	4	bb–1235.Eb02
E12.4	+123456.	5	bbb1235.Eb02
E12.4	–123456.	6	bb–1235.Eb02
D12.4	+123456.	–1	bb0.0123Db07
D12.4	–123456.	0	b–0.1235Db06

OUTPUT FORMAT	INTERNAL NUMBER	SCALE FACTOR	CHARACTERS OUTPUT
D12.4	+123456.	1	bb1.2346Db05
D12.4	−123456.	2	b−12.346Db04
D12.4	+123456.	3	bb123.46Db03
D12.4	−123456.	4	b−1234.6Db02
D12.4	+123456.	5	bb12345.Db01
D12.4	−123456.	6	b−12345.Db01

2. For F output, the internal number is multiplied by 10^n before output conversion.

OUTPUT FORMAT	INTERNAL NUMBER	SCALE FACTOR	CHARACTERS OUTPUT
F8.2	−123.456	−5	bbb−0.00
F8.2	+123.456	−4	bbbb0.01
F8.2	−123.456	−3	bbb−0.12
F8.2	+123.456	−2	bbbb1.23
F8.2	−123.456	−1	bb−12.35
F8.2	+123.456	0	bb123.45
F8.2	−123.456	1	−1234.56
F8.2	+123.456	2	$12345.6
F8.2	−123.456	4	=1234560
F8.2	+123.456	6	$1234560

3. For Gw.d output, the effect of the scale factor is suspended if the number is in the range of 0.1 < 10^d. For values outside this range, the scale factor has the same effect as with E output.

OUTPUT FORMAT	INTERNAL NUMBER	SCALE FACTOR	CHARACTERS OUTPUT
G12.4	+0.012345	2	bbb12.35E−03
G12.4	−0.123456	2	b−0.1235bbbb
G12.4	+1.234567	2	bbb1.235bbbb
G12.4	−1234.56	2	bb−1235.bbbb
G12.4	+12345.6	2	bbb12.35Eb03
G12.4	−123456.0	2	bb−12.35Eb04

4. For I, A, X or H format descriptors, the scale factor has no effect.

Scale Factor Effect On Input

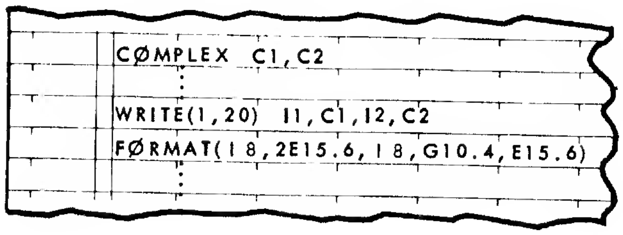
1. For E, F, G, or D format descriptors, the internal value is equal to the external number divided by 10^n (n=current scale factor value). The effect of the scale factor is suspended, however, if the external number contains an E or D scale factor.

INPUT FORMAT	SCALE FACTOR	CHARACTERS INPUT	INTERNAL NUMBER
E12.4	−3	bbbbbb123456	+12345.6
E12.4	−2	−bbbbb123456	−1234.56
E12.4	−1	bbbbb+123456	+123.456
E12.4	0	bbbbb−123456	−12.3456
E12.4	1	bbbbbb123456	+1.23456
E12.4	2	bbb−bb123456	−0.123456
E12.4	3	bbbbbb123456	+0.0123456
E12.4	−3	bbbbbb1234.5	+1234500.0
E12.4	−2	bb1234.5E+01	+12345.0
E12.4	−1	b0.63217E−02	+0.0063217
E12.4	0	bbb−6.7809D2	−678.09

2. For I, A, X or H format descriptors, the scale factor has no effect.

Complex Number Conversion

Complex numbers are made up of two single precision real numbers. Each of the numbers are described using E, F, or G format descriptors.



```
CØMPLEX C1,C2
   :
WRITE(1,20) I1,C1,I2,C2
FØRMAT(I8,2E15.6,I8,G10.4,E15.6)
   :
```

L-Type Conversion (Basic Field Specification Lw or nLw)

The external format of a logical quantity is T or F; and the internal format is +1 (for T) or 0 (for F).

L-Output. If the internal value is 0, an F will be output; otherwise, a T will be output. w-1 leading blanks precede the letter.

OUTPUT FORMAT	INTERNAL DATA	CHARACTERS OUTPUT
L1	0	F
L1	1	T
L4	−2	bbbT
L5	3	bbbbT
L6	0	bbbbbF
L0	1	(NO OUTPUT)

L-Input. Leading blanks are ignored. If the first non-blank character is a T, the internal data is set to 1. If the first non-blank character is an F, the internal data is set to 0. If the first non-blank character is not T or F, set the internal data to 0 and set the error flag to ON (flag sensed by OVERFL function subroutine). The remainder of the external field is scanned and ignored.

OUTPUT FORMAT	DDP-24/124/224 FORTRAN		DDP-116 FORTRAN	
	INTERNAL DATA	CHARACTERS OUTPUT	INTERNAL DATA	CHARACTERS OUTPUT
A1	ABCD	A	AB	A
A3	ABCD	ABC	AB	bAB
A4	ABCD	ABCD	AB	bbAB
A5	ABCD	bABCD	AB	bbbAB

INPUT FORMAT	CHARACTERS INPUT	INTERNAL DATA
L1	F	+0
L2	bT	+1
L3	bFb	+0
L4	TRUE	+1
L8	bbbFALSE	+0
L8	TRUEbbbb	+1
L8	bNEUTRAL	+0 (error)
L8	bbb-TRUE	+0 (error)

A-Type Conversion (Basic Field Specification Aw or nAw)

Four characters of alphanumeric information stored in a computer word make up an alphanumeric item (two characters maximum on the DDP-116, -516 FORTRAN). Any character in this format is accepted as an allowable alphanumeric character.

A-Output. For w greater than 4, w-4 leading blanks are output followed by the four characters of the alphanumeric argument. For w less than 4, the first (left-most) w-characters of the alphanumeric argument are output. This discussion can be made applicable to the DDP-116, -516 by substituting the number 2 in all references to number 4.

A-Input. For w greater than 4, the last four characters are stored internally. For w less than 4, w-4 trailing blanks are added. This discussion can be made applicable to the DDP-116, -516 by substituting the number 2 in all references to the number 4.

H-Field Descriptor (Basic Field Specification nH$a_1a_2a_3$ . . . . . a_n or Ha_1)

The nH descriptor causes Hollerith (alphanumeric) information to be read into or written from the n characters (including blanks) that follow the letter H in the format specification list (characters $a_1a_2a_3$ . . . . . a_n). The value of n must be greater than 0. If no value of n precedes the letter H, a value of n is assumed to be 1.

H-Output. The n-characters following the letter H (including blanks) are output.

FORMAT DESCRIPTOR	CHARACTERS OUTPUT
3HXYZ	XYZ
13H16bNOVb1965,b	16bNOVb1965,b
H3	3
1Hb	b

This field specification provides the basic means for supplying headings on output reports. It is, however, also used for vertical line spacing when outputting on the line printer or typewriter. The first character output determines the line spacing as follows:

CHARACTER	VERTICAL SPACING
Blank	One line
0	Two lines
1	Skip to first line of next page
+	No advance (line printer only)
Others	One line (and output first character)

When outputting on the line printer or typewriter, it is common practice to begin each FORMAT statement with a Hollerith specification that will give the proper vertical spacing. Otherwise, a non-blank character may be inadvertently inserted into the control position with undesirable results.

H-Input. n characters are read from the external device and replace the n characters following the letter H. This function is useful for modifying a part of a heading which will be output later (such as date or run number).

ORIGINAL FORMAT DESCRIPTOR	CHARACTERS INPUT	MODIFIED FORMAT DESCRIPTOR
3HXYZ	ABC	3HABC
11H16bNOVb1965	22bDECb1965	11H22bDECb1965
8HbSMITHbb	bbJONESb	8HbbJONESb

X-Field Descriptor (Basic Specification nX or X)

The value of n is an integer number greater than 0. If the letter X is not preceded by an integer, a value of 1 is assumed for n.

X-Output. n spaces are output.

FORMAT DESCRIPTOR	CHARACTERS OUTPUT
X	b
1X	b
3X	bbb
10X	bbbbbbbbbb

X-Input. The next n-characters are read but ignored. This descriptor is useful in skipping over fields that do not need processing on the input record.

/Record Descriptor

Everywhere a comma is used in a format list, the comma can be replaced by a slash character or a series of slashes to indicate record terminations.

On input, a sequence of n-slashes (n ≥ 1) causes:

1. The remainder of the present record to be ignored.

2. The next n-1 records to be skipped over (n records if slashes occur at end of format statement).

3. The next record to be read if additional data is required to satisfy list items in the READ statement.

On output, a sequence of n-slashes causes:

1. The remainder of the present record to be padded with blanks and outputs the record.

2. The next n-1 blank records to be output (n records if slashes occur at end of format statement).

3. The next record to be written if additional data is required to satisfy list items in the WRITE statement.

FORMAT	NO. OF RECORDS PROCESSED
(I3/4E12.5)	2
(/F10.2,D20.12//I6/6E12.6)	5
(6E12.6/10X,4I10,10X,F20.10///)	5

Repeating of Format Descriptors

Repetition of format descriptors is accomplished at four levels.

Repetition of a Single Descriptor. The integer preceding the H or X descriptors has special meaning, and the H and X descriptors cannot be repeated individually. For all other descriptors, the integer preceding their letter indicates the number of times the descriptor is to be repeated before going on to the next format descriptor in the format statement list. For example:

```
FORMAT (4E10.3,2F12.4,6I4)
            is equivalent to
FORMAT (E10.3,E10.3,E10.3,E10.3,F12.4,
  F12.4,I4,I4,I4,I4,I4,I4)
```

Repetition of a Group of Descriptors. A group of descriptors may be enclosed in parentheses and the opening parentheses preceded by an integer constant greater than 0. This will cause the group of descriptors to be repeated the integer number of times before continuing to the next descriptor within the format statement list. An opening parenthesis not preceded by an integer has an implied repeat count of 1, meaning it is not repeated. For example:

```
FORMAT (I6,2(E10.2,4X,3A3),I5)
            is equivalent to
FORMAT (I6,E10.2,4X,A3,A3,A3,E10.2,
  4X,A3,A3,A3,I5)
```

Repetition of Descriptor Groups. The groups described in the preceding subparagraph may themselves be enclosed in parentheses preceded by an integer repeat count. Parentheses structure with a format descriptor list may not exceed a depth of two (not counting the parentheses around the entire format descriptor list). For example:

```
FORMAT (I2,2(I2,3(D10.4,2X)),I2)
            is equivalent to
FORMAT (I2,I2,D10.4,2X,D10.4,2X,
  D10.4,2X,I2,D10.4,2X,D10.4,2X,D10.4,2X,I2)
```

Repetition After Format List Exhausted. If, after using all the format descriptors in the format statement list, there still remains data to be input or output, the format list is rescanned starting at the opening parenthesis that matches the last closing parenthesis in the format list. The parentheses around the complete format list are not considered unless there are no other parentheses in the list. Any repeat count preceding the start re-scan open parenthesis is included in the re-scan. The following examples illustrate these rules:

a. FORMAT (6E12.4)

b. FORMAT (I6 / (F10.3,4E15.6))

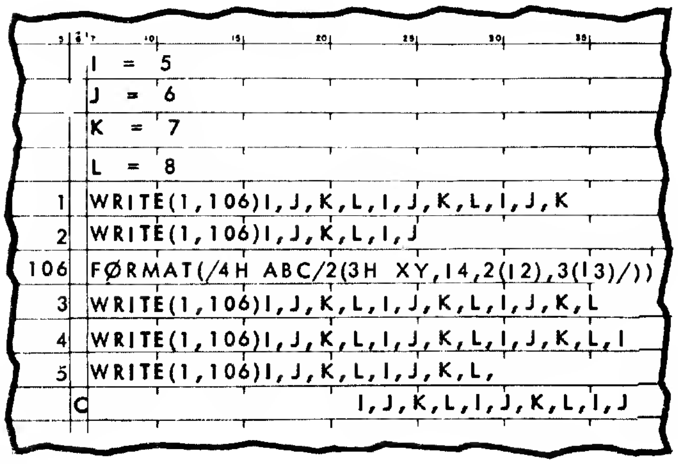
c. FORMAT (I6 / 3(4I5,4X),I5)

d. FORMAT (I6,2(I10,4(I3,2X)))

On output, every time the indicated repetition is made, the current record is padded with blanks and output. A new record is then started. On input, the indicated repetition causes the rest of the current record to be skipped, and the next record to be read.

In example *a*, the first 72 characters of each record are processed. In example *b*, the first 6 characters of the first record are processed, and the first 70 characters of each succeeding record are processed. In example *c*, the first 6 characters of the first record are processed, and the first 77 characters of each succeeding record are processed. Finally, in example *d*, the first 66 characters of the first record are processed, and the first 60 characters of each succeeding record are processed.

Example of Formatted Output. In the following example, the effects of various size data lists, repetition of format descriptors multiple record definitions, and line advance control is shown.

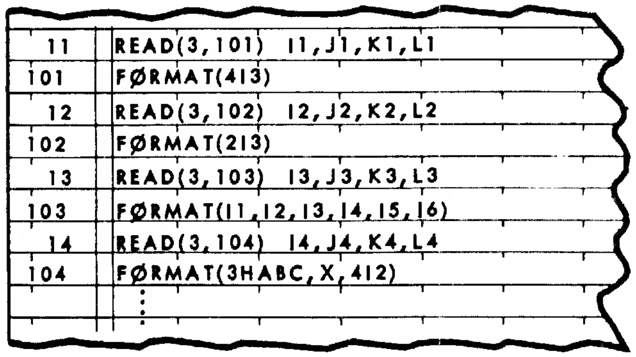


```
      I = 5
      J = 6
      K = 7
      L = 8
    1 WRITE(1,106)I,J,K,L,I,J,K,L,I,J,K
    2 WRITE(1,106)I,J,K,L,I,J
  106 FØRMAT(/4H ABC/2(3H XY,I4,2(I2),3(I3)/))
    3 WRITE(1,106)I,J,K,L,I,J,K,L,I,J,K,L
    4 WRITE(1,106)I,J,K,L,I,J,K,L,I,J,K,L,I
    5 WRITE(1,106)I,J,K,L,I,J,K,L,
     C                      I,J,K,L,I,J,K,L,I,J
```

The following output on a typewriter or line printer would result.

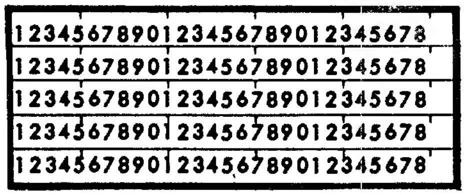
```
ABC
XY   5 6 7   8   5   6  }      Result of statement 1
XY   7 8 5   6   7

ABC
XY   5 6 7   8   5   6  }      Result of statement 2
XY

ABC
XY   5 6 7   8   5   6  }      Result of statement 3
XY   7 8 5   6   7   8

ABC
XY   5 6 7   8   5   6
XY   7 8 5   6   7   8  }      Result of statement 4

XY   5

ABC
XY   5 6 7   8   5   6
XY   7 8 5   6   7   8  }      Result of statement 5

XY   5 6 7   8   5   6
```

Example of Formatted Input. The following example shows the effects of repeating a group of format descriptors when the FORMAT statement does not describe all the arguments in the input statement list. Consider the following FORTRAN statements:



```
   11 READ(3,101) I1,J1,K1,L1
  101 FØRMAT(4I3)
   12 READ(3,102) I2,J2,K2,L2
  102 FØRMAT(2I3)
   13 READ(3,103) I3,J3,K3,L3
  103 FØRMAT(I1,I2,I3,I4,I5,I6)
   14 READ(3,104) I4,J4,K4,L4
  104 FØRMAT(3HABC,X,4I2)
       :
```

If the input data to be read by the preceding statements appeared in the fo''ow.ng form:



```
1234567890123456789012345678
1234567890123456789012345678
1234567890123456789012345678
1234567890123456789012345678
1234567890123456789012345678
```

then:

1. Statement 11 reads one card and defines I1 = 123, J1 = 456, K1 = 789, and L1 = 012.

2. Statement 12 reads two cards. From the first card, it sets I2=123 and J2=456. From the second card, it sets K2=123 and L2=456.

3. Statement 13 reads one card, setting I3=1, J3=23, K3=456, and L3=7890. Format descriptors I5 and I6 are ignored.

4. Statement 14 reads one card, replaces the Hollerith characters ABC in the format statement to the Hollerith characters 123, skips a character of the input data, and sets I4=56, J4=78, K4=90, and L4=12.

FORMAT STATEMENTS ENTERED AT RUN TIME

If the format of BCD data is not known at compile time, it is possible to include the formating data along with the input data.

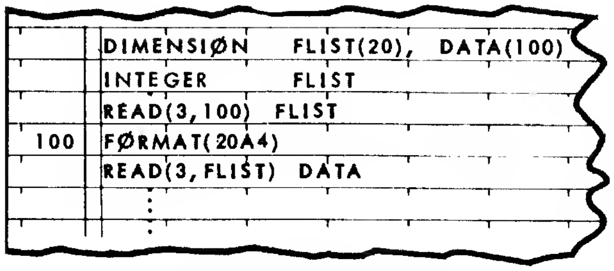
A FORMAT statement, processed at compile time, is stored in BCD format in sequential memory locations. The following chart shows the data generated in memory by the format statement

10 FORMAT (1HO, 3E10.5, 4X, 4I13/).

DDP-24/124/224 (4 char./word)				DDP-116, -516 (2 char./word)	
(	1	H	O	(	1
,	3	E	1	H	O
0	.	5	,	,	3
4	X	,	4	E	1
I	1	3	/	0	.
)				5	,
				4	X
				,	4
				I	1
				3	/
				)	

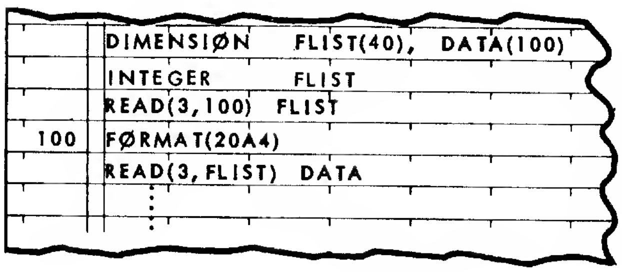
A READ or WRITE statement that refers to an array name instead of a FORMAT statement, is indicating that the format information is to be read into memory to some other input statement (as an alphanumeric array) prior to execution of the READ or WRITE statement. The following examples illustrate the manner in which the list might be entered at run time.

Array FLIST was dimensioned for 20 words since 4 characters are packed per word and there are 80 characters per card. (The DDP-116, -516 FORTRAN packs 2 characters per word; therefore, FLIST is dimensioned for 40 words.) A smaller dimension may be used if the card is not full, or a larger dimension may be used if more than one card is needed to define the format descriptors. FORMAT statement 100 would not have to be adjusted.



```
      DIMENSIØN   FLIST(20),  DATA(100)
      INTEGER     FLIST
      READ(3,100)  FLIST
100   FØRMAT(20A4)
      READ(3,FLIST)  DATA
         :
```

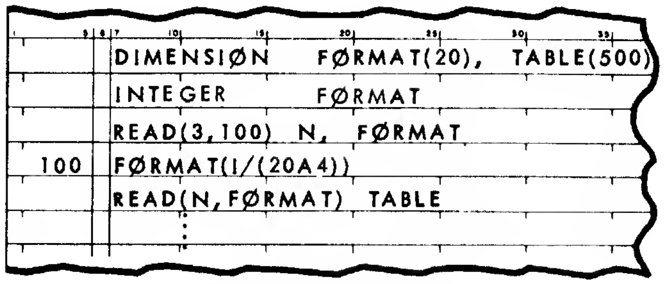
DDP-24/124/224 FORTRAN



```
      DIMENSIØN   FLIST(40),  DATA(100)
      INTEGER     FLIST
      READ(3,100)  FLIST
100   FØRMAT(20A4)
      READ(3,FLIST)  DATA
         :
```

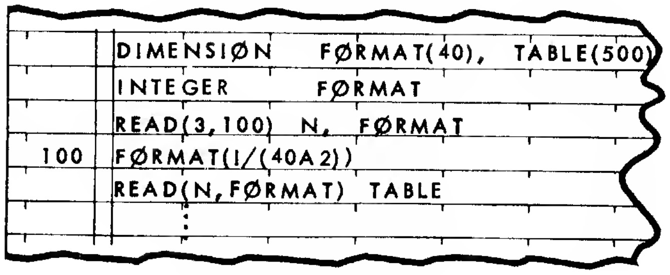
DDP-116, -516 FORTRAN

The following examples have their format lists defined at run time and also refer to the device number symbolically, allowing the device to be determined at run time. This method of defining the device has one drawback in that the subroutine F$RN is unaware in advance which device drivers are required. Therefore the subroutine must load all drivers when the program is loaded.



```
      DIMENSIØN   FØRMAT(20),  TABLE(500)
      INTEGER     FØRMAT
      READ(3,100)  N,  FØRMAT
100   FØRMAT(I/(20A4))
      READ(N,FØRMAT)  TABLE
         :
```

DDP-24/124/224 FORTRAN



```
      DIMENSIØN   FØRMAT(40),  TABLE(500)
      INTEGER     FØRMAT
      READ(3,100)  N,  FØRMAT
100   FØRMAT(I/(40A2))
      READ(N,FØRMAT)  TABLE
         :
```

DDP-116, -516 FORTRAN

SECTION VI
SUBPROGRAMS

Often it is found that a sequence of coding or a sequence of statements used to evaluate a function is required several times within a program, the only difference in the evaluation being the arguments for that function. In such instances, it is desirable to express the function once rather than repeat the coding every time the function is used. This can be done by preparing subprograms which perform commonly recurring operations and keeping them on a library tape for progr.m use at execution time. Another method of eliminating the repetition of coding is to include in the FORTRAN compiler certain basic subprograms which will be compiled as part of the object program.

Sometimes certain subunits of the program are also repeated frequently, and it may be desirable to ›rite each of these once and call for them many .mes. In this case, there is a choice of several types of subprograms.

In addition to eliminating repetitive code, another advantage in constructing programs from subprograms is that it may be possible to use the same "building blocks" in other programs or in modifications to the original program.

In FORTRAN there are five types of subprograms which are:

Library functions
Statement functions
FORTRAN functions
Subroutine subprograms
BLOCK DATA subprogram

Of these, the first three result in a single value and are designated functions; the fourth may result in more than one value and is designated a subroutine; and the fifth is used to initialize data into COMMON areas and is also designated a subroutine.

For each type of subprogram, there are standard ›ractices which must be followed in reference to .alling, naming, and defining (generating) the subprogram.

All functions are incorporated into the object program by referencing in a source program the name of the function in the expression part (right-hand side) of an arithmetic formula. The following are examples of arithmetic expressions including function names:

```
Y = A - SIN (B-C)
C = MIN0 (M,L)
```

The names of library, statement, and FORTRAN functions are used as shown in the example. The appearance of a function name in the arithmetic expression serves to "call" the function. The value of the function is then computed, using the arguments which are supplied in the parentheses following the function name. Only one value is produced by each of the three functions.

FUNCTION SUBPROGRAMS

Function names consist of one to six alphabetic and numeric characters (not special characters), the first of which must be alphabetic. The first character must be I, J, K, L, M, or N (or defined as INTEGER) if the value of the function is to be fixed point. Alternately, a specification statement may be used to type a function value as REAL, INTEGER, DOUBLE PRECISION, COMPLEX, or LOGICAL. The name of the function is followed by parentheses enclosing the arguments (which may be expressions) separated by commas. Some examples of function usage are:

```
SIN (A + B)
SOME (X, Y)
SQRT (SIN (A))
ITAN (3.*X)
```

LIBRARY FUNCTIONS

Library functions are external prewritten functions of a special type and are usually written in assembly language. Initially, they were designed to be used on the library tape—hence the name. These functions are closed subroutines; that is, instead of appearing in the object program each time they are referenced in the source program, they appear only once.

Table 6–1 lists the library functions in the FORTRAN IV library. A more detailed description of the functions is given in the individual subroutine write-ups contained in the library documentation.

TABLE 6—1.

FORTRAN IV LIBRARY FUNCTIONS

FUNCTION NAME	ARGUMENT MODE*	RESULT MODE	FUNCTION DEFINITION
SIN	REAL	REAL	SINE (a) (radians)
DSIN	DOUBLE	DOUBLE	
CSIN	COMPLEX	COMPLEX	
COS	REAL	REAL	COSINE (a) (radians)
DCOS	DOUBLE	DOUBLE	
CCOS	COMPLEX	COMPLEX	
ATAN	REAL	REAL	ARCTANGENT (a)
DATAN	DOUBLE	DOUBLE	
ATAN2	REAL (2)	REAL	ARCTANGENT (a1/a2)
DATAN2	DOUBLE (2)	DOUBLE	
TANH	REAL	REAL	HYPERBOLIC TANGENT (a)
SQRT	REAL	REAL	$\sqrt{a}$
DSQRT	DOUBLE	DOUBLE	
CSQRT	COMPLEX	COMPLEX	
EXP	REAL	REAL	$e^{(a)}$
DEXP	DOUBLE	DOUBLE	
CEXP	COMPLEX	COMPLEX	
ALOG	REAL	REAL	$LOG_e(a)$
DLOG	DOUBLE	DOUBLE	
CLOG	COMPLEX	COMPLEX	
ALOG10	REAL	REAL	$LOG_{10}(a)$
DLOG10	DOUBLE	DOUBLE	
ABS	REAL	REAL	$\|a\|$
IABS	INTEGER	INTEGER	
DABS	DOUBLE	DOUBLE	
CABS	COMPLEX	REAL	$\sqrt{a_r^2 + a_i^2}$
AMOD	REAL (2)	REAL	a_1 (mod a_2)
MOD	INTEGER (2)	INTEGER	
DMOD	DOUBLE (2)	DOUBLE	
AINT	REAL	REAL	Truncate to an integer
INT	REAL	INTEGER	
IDINT	DOUBLE	INTEGER	
AMAX0	INTEGER (>1)	REAL	Choose largest argument
AMAX1	REAL (>1)	REAL	
MAX0	INTEGER (>1)	INTEGER	
MAX1	REAL (>1)	INTEGER	
DMAX1	DOUBLE (>1)	DOUBLE	
AMIN0	INTEGER (>1)	REAL	Choose smallest argument
AMIN1	REAL (>1)	REAL	
MIN0	INTEGER (>1)	INTEGER	
MIN1	REAL (>1)	INTEGER	
DMIN	DOUBLE (>1)	DOUBLE	
FLOAT	INTEGER	REAL	Convert argument mode
IFIX	REAL	INTEGER	
SNGL	DOUBLE	REAL	
REAL	COMPLEX	REAL	
AIMAG	COMPLEX	REAL	
DBLE	REAL	DOUBLE	
CMPLX	REAL (2)	COMPLEX	
SIGN	REAL (2)	REAL	Transfer sign
ISIGN	INTEGER (2)	INTEGER	
DSIGN	DOUBLE (2)	DOUBLE	
DIM	REAL (2)	REAL	Positive difference
IDIM	INTEGER (2)	INTEGER	
CONJG	COMPLEX	COMPLEX	Complex conjugate

*The number in parenthesis specifies the number of arguments.

Notice, as in the example below, that a function may have more than one argument; as in general mathematical usage, multiple arguments are separated by commas.

```
A=SQRT(X)+AMAX1(B1,B2,B3)/SIN(X+0.5)
B=1.0+SQRT(A**2  +  ALØG(CØS(A*3.14)+0.5))
```

STATEMENT FUNCTIONS

Certain functions such as square root, sine, and log can be written as arithmetic expressions. These functions are restricted to those available in the library. It is possible, however, to write expressions involving functions peculiar to the problem at hand. Each desired function is defined by a statement function. For example, if the function

$$g(x) = 1.3 + \sqrt{4.1x + x^2}$$

is to be used several times in a program, a statement function defining G(X) might be written:

```
      GXX(X) = 1.3+SQRT(4.1*X + X**2)
```

An arithmetic formula later employing GXX in the program might be:

```
      Y  =  10.3*  GXX(ALPHA*BETA)+14.7
```

this use of GXX, before the value of the function is computed, the quantity ALPHA*BETA is substituted for X in the expression defining GXX.

In general, statement functions must conform to the following rules:

1. All statement functions must be the first executable statements in that program.

2. The function name must have one to six alphabetic or numeric characters, the first of which must be alphabetic.

3. The name of the function is followed by parentheses enclosing the argument or arguments. Multiple arguments are separated by commas. Each argument must be a single nonsubscripted variable.

4. Any argument that is a real variable in the definition of a function must be a real quantity in any subsequent use of the function. A similar rule applies to arguments of other modes.

5. The value of a function is a real quantity unless the name of the function begins with I, J, K, L, M or N; in which case the value is an INTEGER quantity, or the function may be specified as REAL, INTEGER, DOUBLE PRECISION, COMPLEX, or LOGICAL in a specification statement.

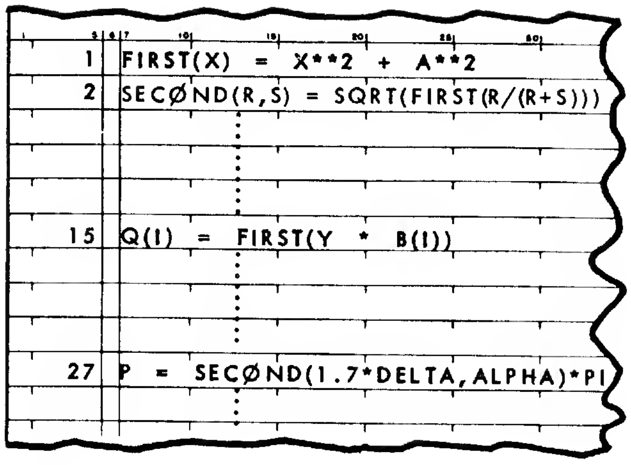
6. The right side of a function statement can be any expression that meets the requirements specified for expressions, except that dummy variables cannot be subscripted. It may involve functions freely, including library functions, previously defined statement functions, and FORTRAN functions.

7. No function can be used as an argument of itself.

8. Any number of variables appearing in the expression on the right side of a function can be stated on the left side as arguments of the function. Since the arguments are only dummy variables, their names are unimportant (except as indicating mode) and may be the same as names appearing elsewhere in the program.

9. Variables on the right side of a function that are not stated as arguments are treated as parameters. The naming of parameters must follow the normal rules of uniqueness.

Typical statement functions are:



```
    1 FIRST(X) = X**2 + A**2
    2 SECØND(R,S) = SQRT(FIRST(R/(R+S)))
              :
              :
              :
   15 Q(I) = FIRST(Y * B(I))
              :
              :
              :
   27 P = SECØND(1.7*DELTA,ALPHA)*PI
              :
```

FORTRAN FUNCTIONS

There are situations in which it is desired to use a particular function in an arithmetic statement, but this function cannot be defined by a single arithmetic statement. However, if this mathematical relationship has a *single result*, the FORTRAN function subprogram may be used. Compiling a FORTRAN function produces a function subprogram in the same form required for a library function. Except for the method generated, FORTRAN and LIBRARY functions are identical in use and format.

FORTRAN functions are closed subprograms not stored within the range of the main program. The main program transfers control to the subprogram as required. After the subprogram has completed the required calculation, control reverts to the main program. Although the subprogram may be used several times in the total structure, it appears in storage only once. A FORTRAN function can be compiled independently of the main program which means that the function can be used with different main programs. One of the primary differences between a FORTRAN function and a subroutine subprogram is that a function returns with a single value while a subroutine may return with multiple values.

The general form in which a FORTRAN function is written is:

FUNCTION NAME (Argument$_1$, Argument$_2$, . . .)
Arithmetic statements to evaluate the function
NAME = Final calculation
RETURN

The FUNCTION statement must be the first statement of the subprogram and defines it as such. The FORTRAN function may consist of many state-

```
      FUNCTIØN   SUM(A,B)
      DIMENSIØN   A(500), B(500)
      SUM = A(1)+B(1)
      DØ 5   J=2,500
    5 SUM = SUM + A(J) + B(J)
      RETURN
      END
$0        END ØF JØB
```

ments of any type except the statements FUNCTION, SUBROUTINE, or BLOCK DATA.

The name of the FORTRAN function consists of one to six alphabetic or numeric characters, the first of which must be alphabetic. The first character must be I, J, K, L, M or N if the value of the function is to be INTEGER; otherwise, the function value is REAL. The mode of the function can also be set by preceding the word FUNCTION with the word INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL; in which case, the first letter mode determination convention is overruled. The function name must not be the same as that of any variable appearing elsewhere in the FORTRAN function or in any program which uses the function.

There must be at least one argument; and there may be as many as required in the subprogram. The arguments must be non-subscripted variable names. If any of the arguments are arrays, a DIMENSION statement involving these arguments is necessary. The arguments may be any variable names occurring in executable statements in the subprogram. Actually, these are dummy variables and the calculation is set up in terms of these dummy variables. A dummy variable in a FORTRAN function must not appear on the left side of an arithmetic statement except as a subscript. The reason is that functions may not change the value of the arguments supplied to the subprogram by the main program. Similarly, a dummy variable must not appear in an ASSIGN statement.

The arithmetic statements to evaluate the function are written in the normal fashion in terms of arguments and constants. The subprograms must evaluate a single-valued function (one which has one and only one value for a given set of arguments). The name of the function must be used as a variable and evaluated by an arithmetic statement; or stated another way, the name of the function must appear on the left side of an arithmetic statement. It is the value of the function name, used as a variable, that is returned as the function value.

A RETURN statement indicates the conclusion of the subprogram, and takes the form:

RETURN

This statement terminates the subprogram and returns control to the main program. A RETURN statement must be the last statement to which control passes in a FORTRAN function; that is, it must be the last statement logically, but it is not necessary that it be last physically. For example:

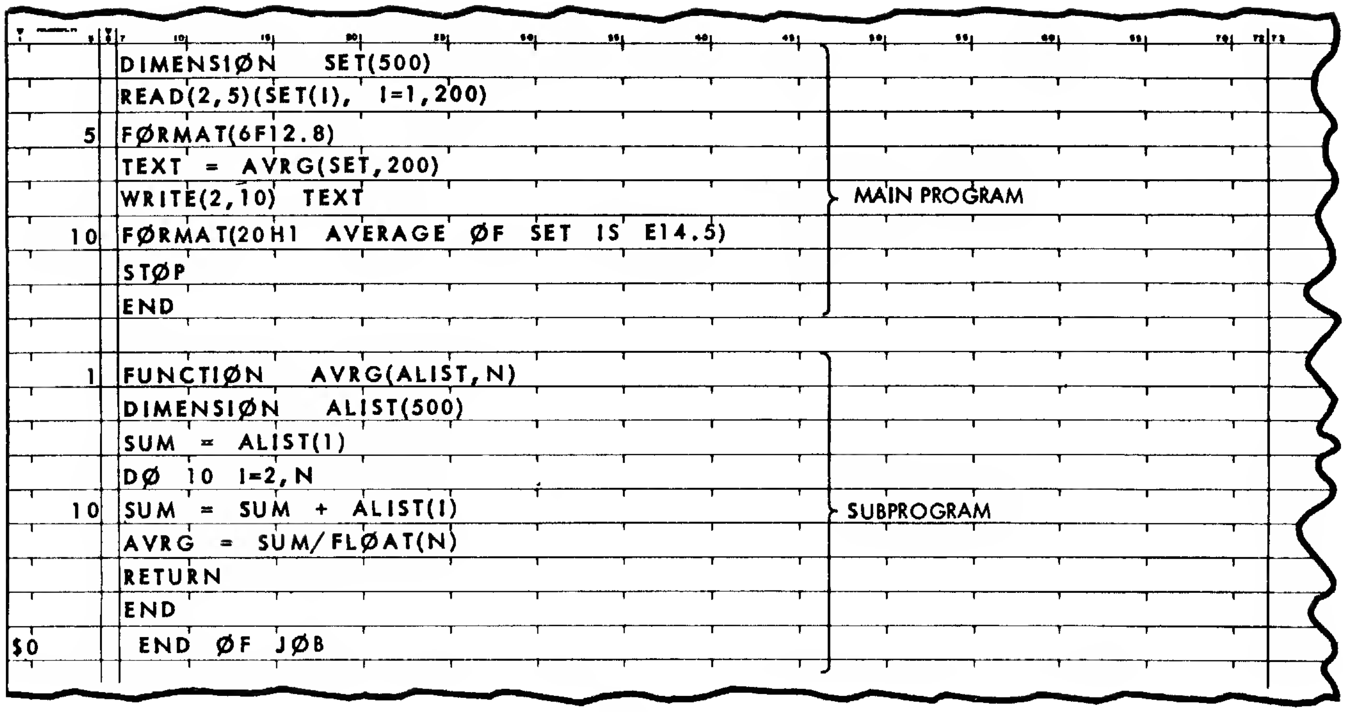
```
      CØMPLEX FUNCTIØN   CARTI(R,THETA)
      X = R * CØS(THETA/57.3)
      Y = R * SIN(THETA/57.3)
      CARTI = CMPLX(X,Y)
      RETURN
      END
$0        END ØF JØB
```

Using a FORTRAN Function Subprogram.

Statement. A subprogram introduced by a FUNCTION statement is called for in the main program by an arithmetic formula involving the function name. For example, as in Figure 6--1, the subprogram introduced by FUNCTION AVRG (ALIST, N) could be called for in the main program by the arithmetic formula:

TEXT = AVRG (SET, 200)

Arguments. The list of arguments in the main program may contain any legitimate FORTRAN constant, variable (subscripted or non-subscripted), expression, subfunction name, or name of any array provided the corresponding dummy variable in the subprogram has the same mode. A Hollerith argument cannot be used. There must be agreement in number, order and mode between the argument list following the function name in the main program and the argument list (dummy variables) in the FUNCTION statement. The subprogram must contain the same DIMENSION statements as the main program. Subfunction names included in argument lists must also appear in an EXTERNAL statement. (See Section IV.)



```
      DIMENSIØN   SET(500)
      READ(2,5)(SET(I),  I=1,200)
    5 FØRMAT(6F12.8)
      TEXT  =  AVRG(SET,200)
      WRITE(2,10)  TEXT                          MAIN PROGRAM
   10 FØRMAT(20H1  AVERAGE  ØF  SET  IS  E14.5)
      STØP
      END

    1 FUNCTIØN   AVRG(ALIST,N)
      DIMENSIØN   ALIST(500)
      SUM  =  ALIST(1)
      DØ  10  I=2,N
   10 SUM  =  SUM  +  ALIST(I)                   SUBPROGRAM
      AVRG  =  SUM/FLØAT(N)
      RETURN
      END
$0      END  ØF  JØB
```

FIGURE 6—1

SUBROUTINE SUBPROGRAMS

Some desirable building blocks have multiple outputs and can be compiled as SUBROUTINE subprograms. Each may also have multiple inputs and the calculation may require many statements.

The SUBROUTINE subprogram is compiled independently of the main program and is called for by a separate statement. When it is desired to use a SUBROUTINE subprogram the main program contains a statement in the form:

CALL NAME (Argument$_1$, Argument$_2$, ...)

Control is transferred at this point to the specified subroutine. When the calculations in the subroutine are finished, control is transferred to the statement following the CALL in the main program.

The general form in which a SUBROUTINE subprogram is written is:

SUBROUTINE NAME (Argument$_1$, Argument$_2$ . .)

Statements to evaluate required results

RETURN

END

The SUBROUTINE statement must be the first statement of the subprogram; it defines it as a subroutine.

The name of a subroutine consists of one to six alphanumeric characters, the first of which is alphabetic. In fact, the subroutine name must not be the same as any variable (subscripted or not) appearing elsewhere in the subprogram, or any program which calls it.

The arguments stated in the subroutine are dummy variables representing input and output variables; each is either a variable name, an array name or a subfunction name. It is allowable to have a subroutine with no arguments. If an argument is the name of an array, it must appear in a DIMENSION statement following the SUBROUTINE statement. The arguments in the

SUBROUTINE list usually contain one or more dummy variables representing the result or results to be returned to the main program.

The intermediate part of the subroutine may contain any of the usual FORTRAN statements, arithmetic, control, input-output, or specification, except the statements FUNCTION, SUBROUTINE, DATA, and BLOCK DATA. The dummy variables representing the results of the subroutine may be used freely on the left side of arithmetic statements. Each dummy variable representing a result must appear at least once on the left side of a statement so that the value will be stored for future use.

A subroutine is terminated by a RETURN statement which is the last statement (logically) to which control passes in a subroutine. The last card physically in each subprogram must be an END card.

Using A Subroutine Subprogram. When it is desired to use a SUBROUTINE subprogram in a program, a CALL statement is used to transfer control to the subroutine. The CALL statement is of the form:

CALL NAME (Argument$_1$, Argument$_2$, . . .)
or
CALL NAME

where NAME represents the symbolic name of a subroutine. The subroutine must be available to the main program at the time of execution of the program.

The arguments may have any one of seven forms:

1. Constant
2. Hollerith constant
3. Variable name
4. Subscripted variable name
5. Array name
6. Expression
7. Subfunction name

A subfunction argument must be defined in the calling program. Except for Hollerith constants, the list of arguments in the CALL statement must agree in number, order, and mode with the list given in the SUBROUTINE statement. If any of the arguments are arrays, equivalent DIMENSION statements must appear in the subroutine and main program.

A Hollerith constant in the argument list consists of an integer number, the letter H, followed by the integer-number of characters. The characters are stored as an array containing four characters per word (two characters per word in the DDP-116, -516) with the last word left-justified if necessary. The equivalent item in the subroutine's argument must be an array of large enough length to hold all characters.

Assume, for example, it is desired to multiply matrix A, N rows and M columns, by matrix B, M rows and L columns; the product matrix C has N rows of L columns. The subroutine shown in Figure 6–2 accomplishes this operation.

A DIMENSION statement following the SUBROUTINE statement specifies the maximum size of the matrices that can be used.

BLOCK DATA SUBPROGRAM

The DATA statement can be used to initialize values of variables or array items (see Section IV) but cannot initialize values of items in blank or labeled COMMON areas. Blank COMMON areas cannot be initialized, but labeled COMMON areas can be initialized by a BLOCK DATA subprogram in which the labeled COMMON areas are first described by specification statements and then initialized by DATA statements. This is the only purpose a BLOCK DATA subprogram can have.

A BLOCK DATA statement takes the form:

BLOCK DATA

This statement may only appear as the first statement of specification subprograms that are called BLOCK DATA subprograms, and that are used to enter initial values into elements of labeled COMMON blocks. This special subprogram contains only type-statements, EQUIVALENCE, DATA, DIMENSION, and COMMON statements. If an entity of a given COMMON block is given an initial value in such a subprogram, a complete set of specification statements for the entire block must be included, even though some of the elements of the block do not appear in DATA statements. Initial values can be entered into more than one block in a single subprogram.

```
    1 DIMENSIØN    X(10,15),  Y(15,12),  Z(10,12),LIST (2)
      READ(2,4)  ((X(I,J),  J=1,15),I=1,10),
     X                ((Y(I,J),J=1,12),I=1,15)
    4 FØRMAT(6E12.6)
    5 CALL  MATMPY(X,5,10,Y,7,Z)
      DØ  13  J=1,7                                    MAIN PROGRAM
   13 WRITE(4,15)  (Z(I,J),I=1,5)
   15 FØRMAT(1H0  6E17.6)
      CALL  EXIT(6HTP1732)
      STØP
      END

    1 SUBRØUTINE    MATMPY(A,N,M,B,L,C)
      DIMENSIØN    A(10,15),B(15,12),C(10,12)
      DØ  5  I=1,N
      DØ  5  J=1,L
    3 C(I,J)  =  0.0                                   SUBROUTINE
      DØ  5  K=1,M
    5 C(I,J)  =  C(I,J)+A(I,K)*B(K,J)
      RETURN
      END

      SUBRØUTINE  EXIT  (LIST)
      DIMENSIØN    LIST(2)
      WRITE(1,5)  LIST
    5 FØRMAT(12H  END  ØF  JØB  ,A8,  /)               SUBROUTINE
      RETURN
      END
$0        END  ØF  JØB
```

FIGURE 6—2

```
      BLØCK  DATA
      CØMMØN  /CØM1/C2,C3,ARR/CØ M2/X,Z,C
      DIMENSIØN    ARR(40)
      EQUIVALENCE(C1,ARR(1)),(C4,ARR(2))
      INTEGER  Z
      CØMPLEX  C
      DATA   C1,C2,C3,C4/4*0.0/,  Z,C/45,(1.3,3.14)/
      END
```

SECTION VII
FORTRAN SYSTEM DESCRIPTION

LISTINGS

Symbolic Listings

The operator may choose to have a symbolic listing generated along with the object output tape during compilation of a FORTRAN program. The method of making this choice and specifying the output device is detailed in the specific FORTRAN compiler operating instructions for each computer. The listing consists of two types of lines: source statement and object output.

Source statement lines are inserted into the listing prior to the object coding that the source statement generates. The line consists of the card image (or images) that make up a source statement.

The object output line consists of a relative location ress (octal), a mnemonic operation code, and an address field. If the address references a variable, it is output in symbolic form. If the address references a constant, the first word of the constant is output in octal form. If the address is absolute, it is output in octal form. If the address is relative (not symbolic), a local address is given. Indexing and indirect indicators are also included when applicable.

A sample of a typical symbolic listing generated by the DDP-24/-124/-224 compiler follows. The DDP-116/-516 compiler generates a similar format but with different instruction mnemonics and shorter octal constants.

```
   7 INTER=ITER+1
0000 LDA ITER
0001 ADD='00000001
0002 STA INTER
       K=1
0003 LDA='00000001
0004 STA K
  22 BIGX=X/Y
0005 LDA X
0006 LDB X
0007 JST D$22
0010 NOP Y
0011 STA BIGX
0012 STB BIGX
C---
       JM1=J-1
0013 LDA J
0014 SUB='00000001
0015 STA JM1
       DO 26 J=3,10
0016 LDA='00000003
0017 STA J
   6 JM1=JM1+J
0020 LDA JM1
0021 ADD J
0022 STA JM1
0023 LDA J
0024 ADD='00000003
0025 SKG='00000012
0026 JMP 70017
```

Error Messages

Any time the compiler detects an error in format of a FORTRAN statement, a two line error message is typed or printed in the listing. If the error is of a type that is recognized as soon as it is encountered, the first line is a duplicate of the line in which the error occurs. If the error cannot be recognized until later in the program, the line at which the error is recognized contains a left pointing arrow (←) in column 6. In either case, the second extra line consists of a row of asterisks broken by the word ERROR in the left-hand margin and the error diagnostic in approximately the same horizontal position as the error. The error diagnostics for the DDP-24/-124/-224 compiler consists of four letter mnemonic whereas the diagnostics for the DDP-116/-516 consists of a two letter mnemonic. Refer to Appendix E for a list of error definitions. Typical error messages for the DDP-116/-516 are:

```
                       K = 3
                    20 A = B**2.0 + K
                    20 A = B**2.0 + K
******** ERROR ********          **** MM ****
                    WRITE (4, 42) A
                    42 FORMAT (F10.4)
```

In statement 20 there is a mixed mode error (MM) which is recognized immediately.

Mapping

Generating a memory map of the program being compiled is divided into two parts. The first part consists of a list of variable names, array names, and constants. This list is generated by the compiler after compilation of a FORTRAN program and under operator option. The method of requesting such a list is detailed in the specific FORTRAN compiler operating instructions for each computer.

The second part of the memory map is generated by the loader program when the object program is loaded into memory. The operating instructions for the appropriate loader outline the procedure for generating this second list. The map generated by the loader consists of three addresses representing the first and last cell of the main program and the entry address for the main program. Following these addresses are the entry addresses of all COMMON blocks and subroutines called by the main program (or its subroutines).

TRACING

Two types of TRACE statements are available for use with the compiler. The first is used in tracing selected variables only, and the second is used in tracing all variables within a specified area.

Item Tracing

A TRACE statement used for item tracing specifies a list of variable names and/or array names. The format for this type of TRACE statement is

TRACE $x_1, x_2, x_3 \ldots . x_m$

where x is any variable or array name. When any of the variables or array elements become re-defined by an arithmetic statement, coding is inserted into the object program, causing a line of trace information to be typed. Such a TRACE statement can be placed anywhere in a program; but insertion of coding for tracing the listed variables does not start until the TRACE statement is processed. As many TRACE statements as desired may be included in the program's source statements.

Area Tracing

A TRACE statement used for area tracing specifies a single statement number and has the format

TRACE n

where n is any statement number not yet defined. This type of TRACE statement inserts coding into the object program that causes the results of all arithmetic expressions (including IF statements) that follow the TRACE statement up to and including the statement specified by n, to output a line of trace information. This group of statements is designated the trace range. In addition to tracing all arithmetic and IF statements within the trace range, all statement numbers defined within the range also cause coding to be generated that outputs a line of trace information, allowing the programmer to follow the sequence of statements as they are executed.

An area TRACE statement should not be placed within the trace range of another area TRACE statement unless all such TRACE statements refer to the same statement number.

Unconditional Trace

If sense switch no. 4 is on during compilation, all arithmetic statements, IF statements, and statement numbers cause coding to be included in the object program for generating lines of trace information at run-time. When the sense switch is reset, trace coding is only generated as directed by the TRACE statement. The sense switch may be set or reset before or during compilation.

Trace Listing Format

At run time of the object program, any trace coding inserted by the compiler causes a line to be typed consisting of a variable name, an array name, or a statement number, followed by an equal sign, followed by the current decimal value assigned to that name. The decimal value is typed in INTEGER, FLOATING POINT, or COMPLEX format. Array names are followed by a subscript indicating the element within the array just modified, as if it were a single dimensioned array. See Figure 7–1 for sample lines of trace information as typed at object run-time.

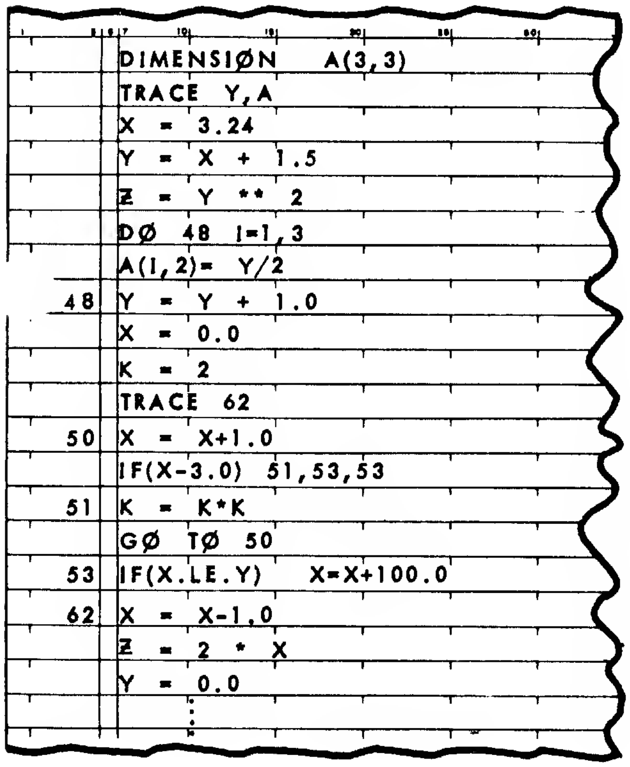
CHAINING FACILITY

The CHAINING facility of FORTRAN IV allows a FORTRAN object program that is too large to fit into the available memory space to be divided into segments. Each segment is run separately and inter-segment communication of data is accomplished through common storage.

```
INTEGER Variable:                          JØKER    =  1739
LOGICAL Variable:                          L1       =  . TRUE.
REAL or DØUBLE PRECISIØN Variable:         DIVERG   =  0. 3217196400E 02
IF Statement Expression Value:             ( )      =  -0. 1934778217E-01
Statement Number:                          (25)
Complex Variable:                          CXDLTA   =  0. 9171978147E 03, 0. 1037200000E 00
```

FIGURE 7—1.

The trace output is inhibited at object program run-time if sense switch no. 4 is set.



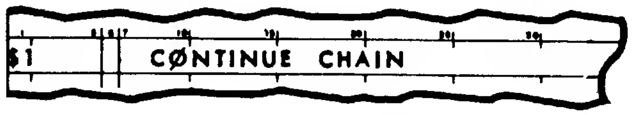
```
      DIMENSIØN    A(3,3)
      TRACE  Y,A
      X  =  3.24
      Y  =  X  +  1.5
      Z  =  Y  **  2
      DØ  48  I=1,3
      A(I,2)=  Y/2
48    Y  =  Y  +  1.0
      X  =  0.0
      K  =  2
      TRACE  62
50    X  =  X+1.0
      IF(X-3.0)  51,53,53
51    K  =  K*K
      GØ  TØ  50
53    IF(X.LE.Y)    X=X+100.0
62    X  =  X-1.0
      Z  =  2  *  X
      Y  =  0.0
        :
```

```
Y       =  0.4740000000E 01
A        (4) = 0.2370000000E 01
Y       =  0.5740000000E 01
A        (5) = 0.2870000000E 01
Y       =  0.6740000000E 01
A        (6) = 0.3370000000E 01
Y       =  0.7740000000E 01
 (50)
X       =  0.1000000000E 01
 ( )    = -0.2000000000E 01
 (51)
K       =        4
 (50)
X       =  0.2000000000E 01
 ( )    = -0.1000000000E 01
 (51)
K       =       16
 (50)
X       =  0.3000000000E 01
 ( )    =  0.0000000000E 00
 (53)
 ( )    =        1
X       =  0.1030000000E 03
 (62)
X       =  0.1020000000E 03
Y       =  0.0000000000E 00
```

Preparing the Source Statements

Three special actions must be taken in order to prepare a program for CHAIN operation:

1. The first card (or line) of all program seg-
nts except the first program segment must be a
cial card with a dollar sign in column 1, the
digit 1 in column 2, and comments in columns
3 through 72.



```
$1      CØNTINUE  CHAIN
```

2. All blank or labeled COMMON areas used for communication between segments of the chain must be declared with a COMMON statement at the beginning of each segment. The declaration

order and size of each area must be identical in each chain segment.

3. When the program is ready to enter the next segment in the program chain, a CALL CHAIN statement is executed. This subroutine causes the next segment to be loaded into memory and automatically started. (Several CALL CHAIN statements may be present in one program segment for convenience.)

Figure 7–2 is an example of a three segment program CHAIN.

Compiling the Source Statements

Each segment of the chain is compiled separately, resulting in three-object tapes.

Preparing the Binary Chain Tape

1. Mount the object tape for segment one and load it into memory using the standard loader and standard loading procedures.

2. When the object tape is loaded, the loader types "MORE" (DDP-24/-124/-224) or "MR" (DDP-116/-516) if subroutines are required. Mount the library tape and press the START button to load the needed subroutines.

3. When all needed subroutines are loaded, the loader types "DONE" (DDP-24/-124/-224) or "LC" (DDP-116/-516).

4. Punch out segment one using the chain dump program. (See appropriate program documentation for details.)

5. Repeat steps 1, 2, 3, and 4 for each additional segment in the chain program. One long binary tape representing the entire chain program will be generated.

Running the Chain Program

1. Mount the special binary tape. After the first segment has been loaded it will be executed automatically.

2. When the first segment transfers to the CALL CHAIN subroutine, the subroutine causes the next segment to be read from the special binary tape and executed. This operation is automatic and no operator action is required.

3. Step 2 is repeated automatically until the last segment is loaded and executed.

Intercommunication Between Segments

All intercommunication between segments is done through variables or arrays in blank or labeled COMMON. Because of the way FORTRAN allocates storage, all blank and labeled COMMON areas that are defined at the beginning of each segment are allocated space at the beginning of that program area.

Note that blank COMMON overwrites the loader as much as posssible since variables of array elements in blank COMMON cannot be initialized with data. Because variables or array elements in labeled COMMON may be initialized, labeled COMMON cannot overwrite the loader on the 24-bit computers. On the 16-bit computers it is possible to overwrite the loader with labeled COMMON. This can be avoided by protecting the loader with a dummy blank COMMON block.

Because the programmer is required to define the size and order of common variables or arrays identical in each chain segment, a reference to a common element in one segment refers to the same memory cells as a reference to the same common element in a different segment. Therefore, communication between segments of a chain program is effected.

OPERATION DETAILS

FORTRAN operating procedures are not given in this manual because the procedures are different for each DDP computer. The operating procedures are provided separately for each computer and include:

Loading Procedure
Sense Switch Settings
Input Formats
Output formats
Listing Formats
Preparing Source Statements
Loading the Object Program
Running the Object Program Data
Error Code Definitions
Estimating Memory Requirements
Memory Maps
Etc.

```
C        CHAIN JØB, SEGMENT NØ. 1
        CØMMØN   A,B,C/CØM2/X,Y,Z,ARRAY
        DIMENSIØN   ARRAY(100)
        READ(3,20)  ARRAY,I,DELTA
```

```
        X = Q1-Q2/Q3
        CALL  CHAIN
        END
$0        END ØF JØB
```

```
C        CHAIN JØB, SEGMENT NØ. 2
$1        CØNTINUE CHAIN
        CØMMØN   A,B,C/CØM2/X,Y,Z,ARRAY
        DIMENSIØN   ARRAY(100),  TABLE(30)
        ALPHA  =  (X-DELTA)/(X+DELTA)*3.1457
```

```
        IF(TEST)10,13,10
10      CALL  CHAIN
13      X  =  TABLE(J)  +  X
        Y  =  TABLE(I)  +  Y
        CALL  CHAIN
        END
$0        END  ØF  JØB
```

```
C        CHAIN JØB,SEGMENT NØ. 3
$1        CØNTINUE CHAIN
        CØMMØN   A,B,C/CØM2/X,Y,Z,ARRAY
        DIMENSIØN   ARRAY(100),XLIST(4,4)
        T=(X+0.5)/(Y+0.5)
```

```
        WRITE(5,14)  ARRAY,XLIST
14      FØRMAT(1H  ,2E10.3)
        STØP
        END
$0        END  ØF  JØB
```

FIGURE 7—2. THREE SEGMENT CHAIN PROGRAM

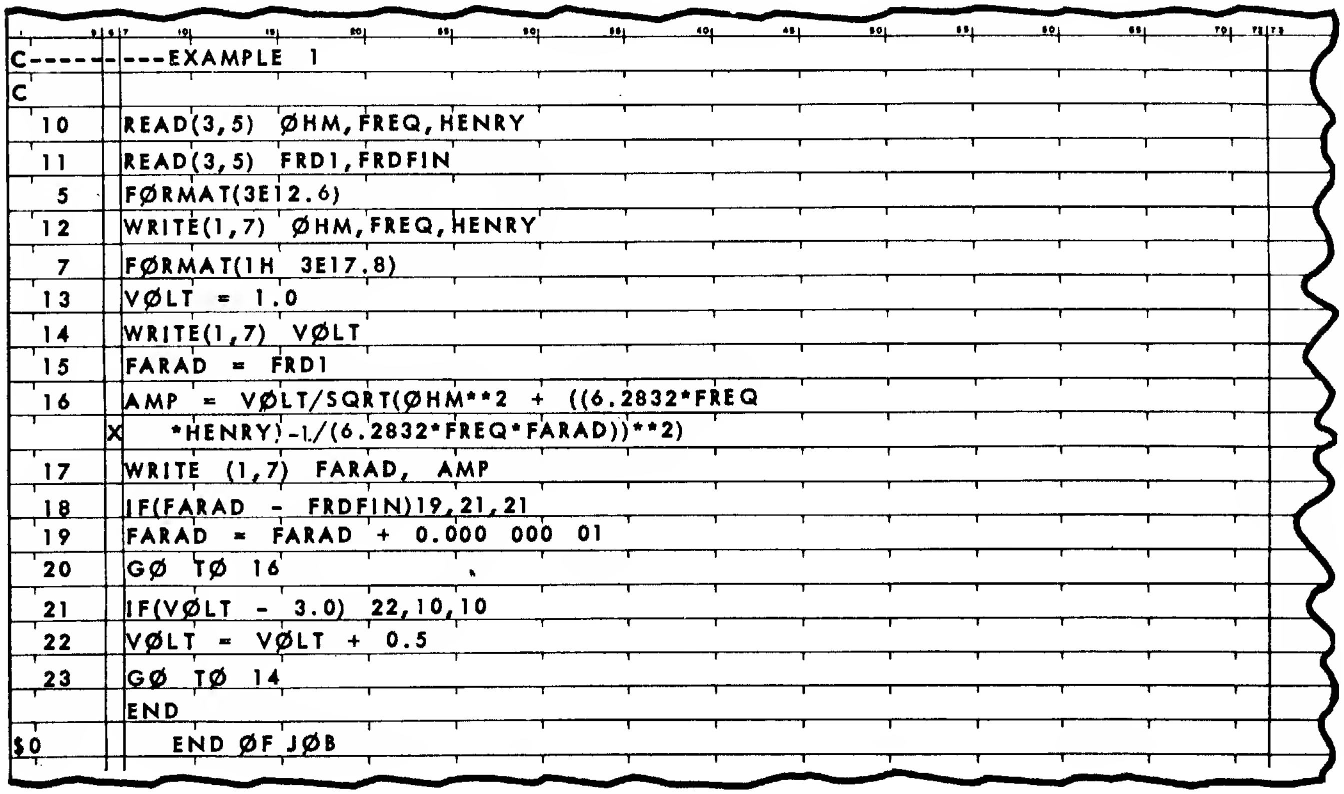
APPENDIX A
SAMPLE PROGRAMS

EXAMPLE 1

The equation for determining the current flowing through an alternating current circuit is:

$$I = \frac{E}{\sqrt{R^2 + \left(2\pi fL - \frac{1}{2\pi fC}\right)2}}$$

The current is to be determined for a number of equally-spaced values of the capacitance (which lie between specified limits) for voltages of 1.0, 1.5, 2.0, 2.5, and 3.0 volts.



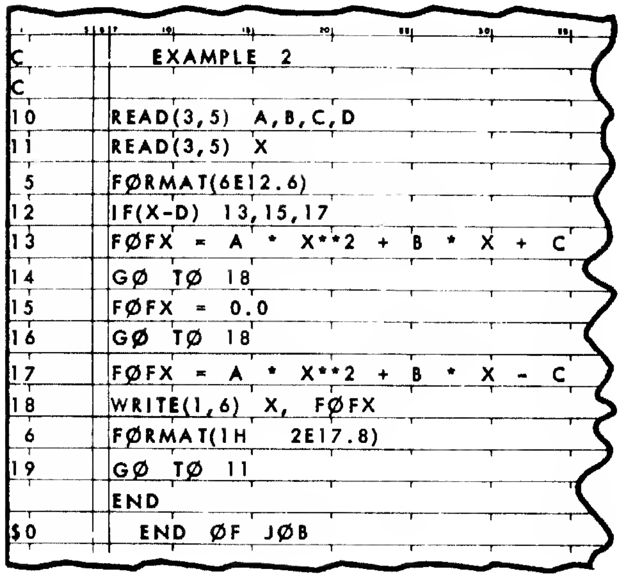
```
C------ ---EXAMPLE 1
C
   10    READ(3,5)  ØHM,FREQ,HENRY
   11    READ(3,5)  FRD1,FRDFIN
    5    FØRMAT(3E12.6)
   12    WRITE(1,7)  ØHM,FREQ,HENRY
    7    FØRMAT(1H  3E17.8)
   13    VØLT  =  1.0
   14    WRITE(1,7)  VØLT
   15    FARAD  =  FRD1
   16    AMP  =  VØLT/SQRT(ØHM**2  +  ((6.2832*FREQ
        X   *HENRY)-1./(6.2832*FREQ*FARAD))**2)
   17    WRITE  (1,7)  FARAD,  AMP
   18    IF(FARAD  -  FRDFIN)19,21,21
   19    FARAD  =  FARAD  +  0.000  000  01
   20    GØ  TØ  16
   21    IF(VØLT  -  3.0)  22,10,10
   22    VØLT  =  VØLT  +  0.5
   23    GØ  TØ  14
         END
$0          END  ØF  JØB
```

EXAMPLE 2

Given values a, b, c, and d punched on cards followed by a set of values for the variable x punched one per card, evaluate the function defined by

$$f(x) = \begin{cases} ax^2 + bx + c & \text{if } x < d \\ 0 & \text{if } x = d \\ -ax^2 + bx - c & \text{if } x > d \end{cases}$$

for each value of x, and type x and f(x).



```
C        EXAMPLE 2
C
10       READ(3,5)  A,B,C,D
11       READ(3,5)  X
 5       FØRMAT(6E12.6)
12       IF(X-D)  13,15,17
13       FØFX  =  A  *  X**2  +  B  *  X  +  C
14       GØ  TØ  18
15       FØFX  =  0.0
16       GØ  TØ  18
17       FØFX  =  A  *  X**2  +  B  *  X  -  C
18       WRITE(1,6)  X,  FØFX
 6       FØRMAT(1H    2E17.8)
19       GØ  TØ  11
         END
$0         END  ØF  JØB
```

EXAMPLE 3

Given:

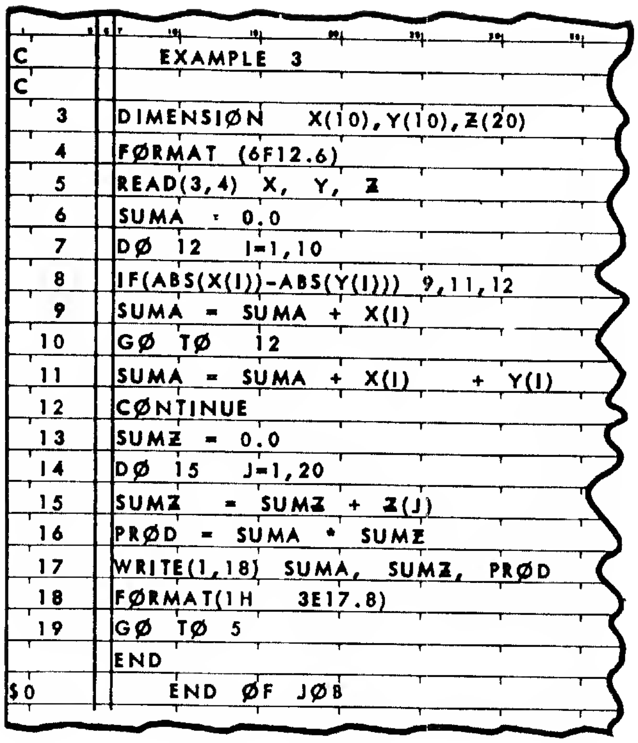
X_i, Y_i, Z_j for $i = 1, \ldots .10$, and $j = 1, \ldots .20$

Compute:

$$\text{PROD} = \left(\sum_{i=1}^{i=10} A_i \right) * \left(\sum_{j=1}^{i=20} Z_j \right)$$

Where:

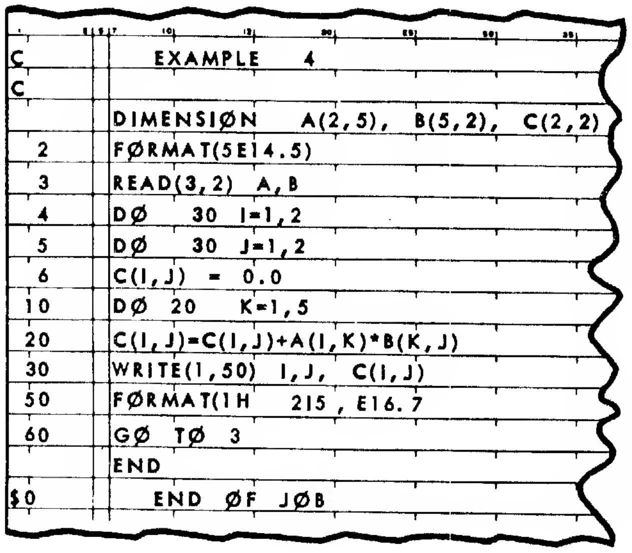
$$\begin{array}{lll} A_i = x_i^2 + Y_i & \text{if} & |X_i| < |Y_i| \\ A_i = x_i + Y_i & \text{if} & |X_i| = |Y_i| \\ A_i = 0 & \text{if} & |X_i| > |Y_i| \end{array}$$



```
C        EXAMPLE 3
C
 3       DIMENSIØN    X(10),Y(10),Z(20)
 4       FØRMAT  (6F12.6)
 5       READ(3,4)  X,  Y,  Z
 6       SUMA  =  0.0
 7       DØ  12    I=1,10
 8       IF(ABS(X(I))-ABS(Y(I)))  9,11,12
 9       SUMA  =  SUMA  +  X(I)
10       GØ  TØ   12
11       SUMA  =  SUMA  +  X(I)     +  Y(I)
12       CØNTINUE
13       SUMZ  =  0.0
14       DØ  15    J=1,20
15       SUMZ   =  SUMZ  +  Z(J)
16       PRØD  =  SUMA  *  SUMZ
17       WRITE(1,18)  SUMA,  SUMZ,  PRØD
18       FØRMAT(1H    3E17.8)
19       GØ  TØ  5
         END
$0         END  ØF  JØB
```

EXAMPLE 4

The following example of matrix multiplication illustrates DO nests and multiple subscripts. (A DO nest is a set of two or more DO statements, the range of one of which includes the ranges of the others.)



```
C        EXAMPLE   4
C
         DIMENSIØN   A(2,5),  B(5,2),  C(2,2)
 2       FØRMAT(5E14.5)
 3       READ(3,2)  A,B
 4       DØ    30  I=1,2
 5       DØ    30  J=1,2
 6       C(I,J)  =  0.0
10       DØ  20    K=1,5
20       C(I,J)=C(I,J)+A(I,K)*B(K,J)
30       WRITE(1,50)  I,J,  C(I,J)
50       FØRMAT(1H    2I5 , E16.7
60       GØ  TØ  3
         END
$0         END  ØF  JØB
```

EXAMPLE 5

Problem:

Compute the following quantities:

$$P_i = \sqrt{\sin^2 (A_i B_i + C_i) + \cos^2 (A_i B_i - C_i)}$$

$$Q_i = \sin^2 (A_i + C_i) + \cos^2 (A_i - C_i)$$

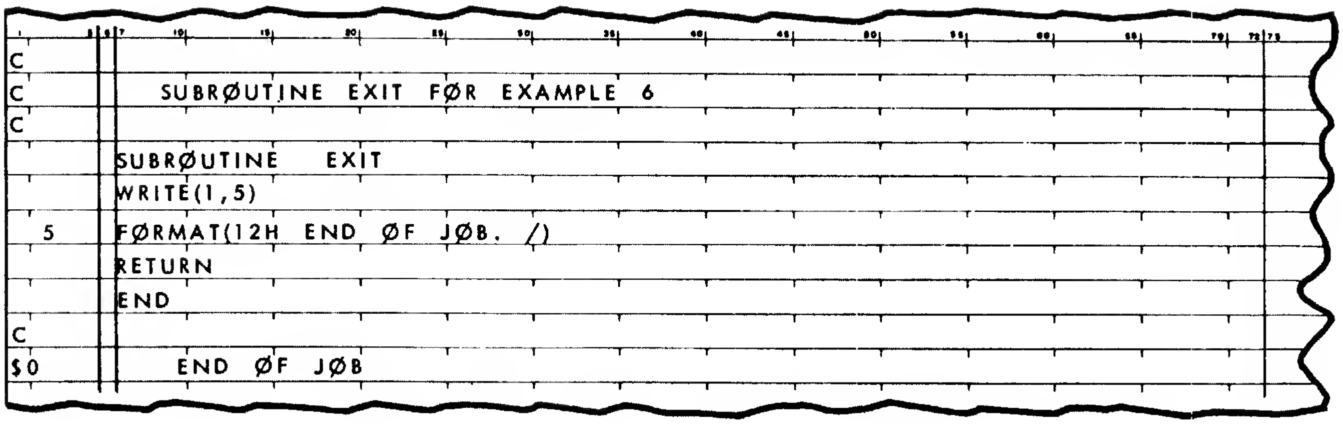
```
C      EXAMPLE  5
C
      DIMENSIØN    A(10),B(10),C(10),
     X                      P(10),Q(10)
      TRIGF(X,Y)=SIN(X+Y)**2 + CØS(X-Y)**2
      READ(3,4)  A,  B,  C
    4 FØRMAT(3F12.6)
      DØ    7    I=1,10
      P(I)=SQRT(TRIGF(A(I)*B(I),C(I)))
    7 Q(I)=TRIGF(A(I),C(I))
      WRITE(1,9)  (A(I),B(I),C(I),P(I),
     X                         Q(I),  I=1,10)
    9 FØRMAT(1H    5F17.4)
      STØP
      END
$0        END  ØF  JØB
```

EXAMPLE 6

Problem:

Find and print two product matrices.

```
C         EXAMPLE    6
C
      DIMENSIØN    X(10,15),  Y(15,12),  Z(10,12),
     2                       D(10,15),  E(15,12),  F(10,12)
      READ(3,4)  ((X(I,J),  J=1,3),  I=1,3),
     1                      ((Y(I,J),  J=1,3),  I=1,3)
    4 FØRMAT(6E12.6)
      CALL  MATMPY(X,3,3,Y,3,Z)
      READ(3,4)  ((D(I,J),J=1,3),  I=1,3),
     X                      ((E(I,J),J=1,3),  I=1,3)
      CALL  MATMPY(D,3,3,E,3,F)
      DØ    13    J=1,3
   13 WRITE(4,15)  (Z(I,J),  I=1,3)
      DØ  14    J=1,3
   14 WRITE(4,15)  (F(I,J),  I=1,3)
   15 FØRMAT(1H  6E17.6)
      CALL  EXIT
      STØP
      END
C
C        SUBRØUTINE  MATMPY  FØR  EXAMPLE  6
C
      SUBRØUTINE    MATMPY(A,N,M,B,L,C)
      DIMENSIØN    A(10,15),  B(15,12),  C(10,12)
      DØ    5    I=1,N
      DØ    5    J=1,L
      C(I,J)  =  0.0
      DØ    5    K=1,M
    5 C(I,J)  =  C(I,J)  +  A(I,K)*B(K,J)
      RETURN
      END
```



```
C
C      SUBRØUTINE EXIT FØR EXAMPLE 6
C
      SUBRØUTINE  EXIT
      WRITE(1,5)
    5 FØRMAT(12H END ØF JØB. /)
      RETURN
      END
C
$0         END ØF JØB
```

APPENDIX B

MODIFICATION OF INPUT/OUTPUT DEVICE ASSIGNMENTS

The device numbers referred to by input or output statements have been tentatively set to the most common devices. Each statement generates a CALL to library subroutine F$Rn, F$Wn, or F$Cn, where n is the device number 0 through 9. The devices referred to can be easily changed by writing different subroutines with the same name, and substituting the new subroutine for the existing subroutine on the library tape. For example, assume F$R2 is used to read paper tape and no paper tape reader is available, but six magnetic tape units are available. A magnetic tape subroutine (similar to F$R5) can be written, named F$R2, and put on the library tape in place of the original F$R2 subroutine. All references to device 2 will now refer to magnetic tape 6 instead of the paper tape reader.

If the unit number is referred to symbolically instead of as an integer constant, a different subroutine calling sequence is generated.

```
CALL F$RN
PZE n
```

F$RN has a 10-place table which interprets n as one of the standard device assignments (0 through 9) and transfers to the proper subroutine. If desired, the F$RN subroutine can be modified to rearrange the device assignment table or to expand it for additional devices. The table can be expanded to any size less than 15000. Therefore, by expanding the table in F$RN, device 10 could be magnetic tape 6, device 11 might be typewriter 2, etc. Only symbolic device numbers can exceed 9 in value. Constant device number calling sequences do not use the F$RN subroutine and do not, therefore, use the device assignment table.

Finally, if a symbolic device number reference is made, but the number (such as n=0 or n=13) is not in F$RN's device assignment table, a message is typed and followed by a halt. The operator can, at that time, set an acceptable device number into the accumulator (right-justified) and press the computer's START button. If the new number is acceptable, input or output is generated using the new device number. However, each time the unacceptable device number is referred to by another READ, WRITE, or Control statement, the typed message followed by a halt occurs again.

APPENDIX C
DYNAMIC STORAGE ALLOCATION

A special version of the FORTRAN compiler and FORTRAN library is provided (only for a DDP-224 computer with 8K of memory, 3 index registers, and hardware floating-point options) when dynamic allocation of storage is desirable. Programs compiled by this version of the compiler, or subroutines on this version of the library are written in a format that causes all variables or temporary storages to be made into a COMMON area and thereby shared where possible with other subroutine temporary storages. This procedure permits (1) shared variable or temporary storages among subroutines, (2) recursive calling of subroutines, and (3) real-time interrupt capability.

SHARED STORAGE

The variable storage required by a subroutine is not defined as part of the subroutine, but as relative to an address in index register 2. Since this index register address is not assigned until the subroutine is actually entered, the storage area needed by the subroutine is shared by all other subroutines that are not called on by this subroutine (subroutines on the same level). Since the address allocation is done at run time and only as needed, dynamic allocation of variable storage is effected by all subroutines written in the dynamic format.

RECURSIVE CALLS

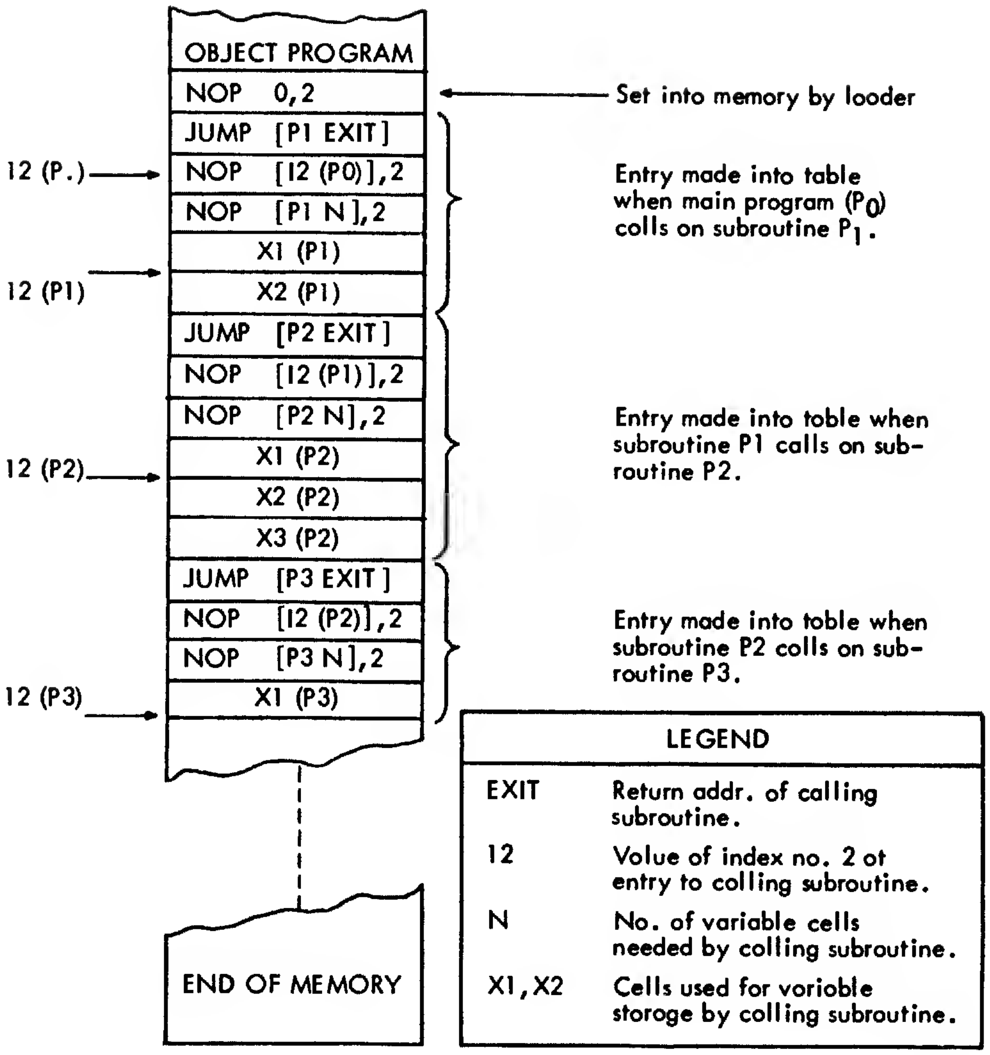
The F$DA subroutine that is called on by all subroutines in the Dynamic Allocation Library, has the current value of index register 2 in a push-down list and assigns a variable storage area independent of the subroutine itself. Therefore, a subroutine written in dynamic format can call itself to any reasonable depth since each depth carries its own return address and variable (temporary storage) list.

REAL TIME INTERRUPT CAPABILITY

A subroutine written in this format may be interrupted by a real-time external interrupt signal. The processor for the interrupt can call upon the same subroutine interrupted, and, upon completion of the interrupt processing, can return to the interrupt point in the subroutine and continue. In fact, if the interrupt processor program is written in this format, it can be interrupted itself (after doing a few set-up instructions) by the same or a different interrupt. This interruption can continue to a depth limited only by the memory size. All the interrupts would be processed eventually on a last-in/first-out basis.

DYNAMIC CONTROL SUBROUTINE

The F$DA subroutine generates a list that consists of three control parameters plus the cells needed for variables by the calling subprogram. The three parameters stored in this list consist of the calling subprogram's return address, the number of cells needed for variable storage by the calling subprogram, and the previous contents of index register 2. (Index register 2 is initially set to L_0.) The format of this table is shown in Figure C-1.


OBJECT PROGRAM
NOP 0,2
JUMP [P1 EXIT]
NOP [I2 (P0)],2
NOP [P1 N],2
X1 (P1)
X2 (P1)
JUMP [P2 EXIT]
NOP [I2 (P1)],2
NOP [P2 N],2
X1 (P2)
X2 (P2)
X3 (P2)
JUMP [P3 EXIT]
NOP [I2 (P2)],2
NOP [P3 N],2
X1 (P3)
END OF MEMORY
I2 (P.)
I2 (P1)
I2 (P2)
I2 (P3)
Set into memory by looder
Entry made into table when main program (P_0) colls on subroutine P_1.
Entry made into toble when subroutine P1 calls on subroutine P2.
Entry made into toble when subroutine P2 colls on subroutine P3.

LEGEND	
EXIT	Return addr. of calling subroutine.
I2	Volue of index no. 2 ot entry to colling subroutine.
N	No. of variable cells needed by colling subroutine.
X1,X2	Cells used for vorioble storoge by colling subroutine.

FIGURE C—1. LIST GENERATED BY F$DA SUBROUTINE

APPENDIX D

STATEMENT, LIBRARY FUNCTION, AND INPUT/OUTPUT SUMMARIES

STATEMENT SUMMARY

CONTROL STATEMENTS

General Form of Statement	*Example*	*Page*
GØ TØ n	GØ TØ 314	3-1
ASSIGN K TØ I	ASSIGN 320 TØ I	3-1
GØ TØ I, $(k_1,k_2,\ldots,k_n)$	GØ TØ I, (100, 310, 320, 409)	3-1
GØ TØ $(k_1,k_2,\ldots,k_n)$, I	GØ TØ (100, 310, 320, 409), I	3-1
IF (e) k_1,k_2,k_3	IF (I + 46) 30, 30, 32	3-2
IF (e) S	IF (L1. ØR. L2) GØ TØ 20	3-2
DØ n I = m_1,m_2,m_3	DØ 20 I = 1,14,2	3-3
DØ n I = m_1,m_2	DØ 11 I = 1,10	3-3
CØNTINUE	CØNTINUE	3-4
PAUSE n	PAUSE 3	3-4
PAUSE	PAUSE	3-4
STØP n	STØP 3	3-5
STØP	STØP	3-5
END	END	3-5
$0	$0 END ØF JØB	1-2
$1	$1 CØNTINUE CHAIN	1-2

SPECIFICATION STATEMENTS

General Form of Statement	*Example*	*Page*
INTEGER $a_1,a_2,\ldots,a_n$	INTEGER A, B, X(10)	4-1
REAL $a_1,a_2,\ldots,a_n$	REAL I,J,K (4,3)	4-1
DØUBLE PRECISIØN $a_1,a_2,\ldots,a_n$	DØUBLE PRECISIØN X, Y, D(10)	4-1
CØMPLEX $a_1,a_2,\ldots,a_n$	CØMPLEX TEST, C2 (3,7)	4-1
LØGICAL $a_1,a_2,\ldots,a_n$	LØGICAL L1,L2 BØØL, L (4,4,4)	4-1
DIMENSIØN $a_1\ (i_1),\ldots,a_n(i_n)$	DIMENSIØN LIST (400), TABLE (10,10,4)	4-2
EXTERNAL $a_1,a_2,\ldots,a_n$	EXTERNAL TEST1,TEST2,TEST3	4-2
EQUIVALENCE $(k_1),(k_2),\ldots,(k_n)$	EQUIVALENCE (X,D1,C3), (A(4,1), D2(3),C1)	4-3
CØMMØN $a_1,a_2,\ldots,a_n$	CØMMØN A,B,C(10)	4-3
CØMMØN $/x_1/a_1/x_2/a_2\cdots$ $\cdots/x_n/a_n$	CØMMØN/LABEL1/D,E,F(6)... .../LABEL2/P(100,10),S;T	4-3
CØMMØN $//a_1,a_2,\ldots,a_n$	CØMMØN //U,G,F,H(4,3)	4-3
DATA $k_1/d_1/;h_2/d_2/,\ldots k_n/d_n/$	DATA A1(4),X,I/0.107,1.0E5, 3/,R/0./	4-4
TRACE $x_1,x_2,\ldots x_n$	TRACE Y,A,B	4-8
TRACE n	TRACE 62	4-8

INPUT/OUTPUT STATEMENTS

General Form of Statement	*Example*	*Page*
READ (u,f) list	READ (3,20)A,B,C(4),ARRAY	5-1
READ (u) list	READ (3) A,B,C(4),ARRAY	5-1
WRITE (u,f) list	WRITE (4,30) (A(I),B(I),I=1,10)	5-1
WRITE (u) list	WRITE (7) A,B,C	5-1
REWIND u	REWIND 7	5-1
BACKSPACE u	BACKSPACE 7	5-1
END FILE u	END FILE 7	5-1
FØRMAT $(s_1,s_2,\ldots,s_n)$	FØRMAT (1H,13,2E10.4)	5-3

SUBPROGRAM STATEMENTS

General Form of Statement	*Example*	*Page*
CALL Name $(a_1,a_2,\ldots,a_n)$	CALL MATMPY (X,I,TABLE)	6-5
CALL Name	CALL EXIT	6-6
FUNCTIØN Name $(a_1,a_2,\ldots,a_n)$	FUNCTIØN SUM (LIST, SIGMA)	6-3
SUBRØUTINE Name $(a_1,a_2,\ldots a_n)$	SUBRØUTINE MATMPY (A,N,ARRAY)	6-5
SUBRØUTINE Name	SUBRØUTINE EXIT	6-5
RETURN	RETURN	6-4

SUMMARY OF INPUT/OUTPUT

List Items

VARIABLES	A,B
SUBSCRIPTED VARIABLES	X(3), Y(I+3,J)
ARRAY NAMES	X,Y
IMPLIED DO-LOOP CONTROL	((X(I),Y(I,J),I-1,8),J-1,4)

Device Assignments (u)

0 = space	5 = Magnetic tape no. 1
1 = Typewriter	6 = Magnetic tape no. 2
2 = Paper tape	7 = Magnetic tape no. 3
3 = Cards	8 = Magnetic tape no. 4
4 = Line printer	9 = Magnetic tape no. 5

Format Descriptors

nP	Scale factor
nX	Skip n characters
nFw.d	REAL value in mixed number format
nEw.d	REAL value in scaled number format
nGw.d	REAL value in mixed or scaled format (by range)
nDw.d	DOUBLE PRECISION value in scaled format
nIw	INTEGER value in integer format
nLw	LOGICAL value as a T or F (TRUE or FALSE)
nAw	REAL value in alphanumeric packed format
nH	Hollerith data for headings or labels
/1 . . . n	Record delimiters for multiple records

Line Spacing Control for Typewriter and Printer

Blank	One line
0	Two lines
1	Skip to fist line of next page
+	No advance (line printer only)
Others	One line (and output first character)

SUMMARY OF LIBRARY FUNCTIONS

FUNCTION NAME	ARGUMENT MODE	RESULT MODE	FUNCTION DEFINITION
SIN	REAL	REAL	SINE (a) (radians)
DSIN	DOUBLE	DOUBLE	
CSIN	COMPLEX	COMPLEX	
COS	REAL	REAL	COSINE (a) (radians)
DCOS	DOUBLE	DOUBLE	
CCOS	COMPLEX	COMPLEX	
ATAN	REAL	REAL	ARCTANGENT (a)
DATAN	DOUBLE	DOUBLE	
ATAN2	REAL (2)	REAL	ARCTANGENT (a1/a2)
DATAN2	DOUBLE (2)	DOUBLE	
TANH	REAL	REAL	HYPERBOLIC TANGENT (a)
SQRT	REAL	REAL	$\sqrt{a}$
DSQRT	DOUBLE	DOUBLE	
CSQRT	COMPLEX	COMPLEX	
EXP	REAL	REAL	$e^{(a)}$
DEXP	DOUBLE	DOUBLE	
CEXP	COMPLEX	COMPLEX	
ALOG	REAL	REAL	LOG_e (a)
DLOG	DOUBLE	DOUBLE	
CLOG	COMPLEX	COMPLEX	
ALOG10	REAL	REAL	LOG_{10} (a)
DLOG10	DOUBLE	DOUBLE	

FUNCTION NAME	ARGUMENT MODE	RESULT MODE	FUNCTION DEFINITION
ABS	REAL	REAL	$\lvert a \rvert$
IABS	INTEGER	INTEGER	
DABS	DOUBLE	DOUBLE	
CABS	COMPLEX	REAL	$\sqrt{a_r^2 + a_i^2}$
AMOD	REAL (2)	REAL	a_1 (mod a_2)
MOD	INTEGER (2)	INTEGER	
DMOD	DOUBLE (2)	DOUBLE	
AINT	REAL	REAL	Truncate to an integer
INT	REAL	INTEGER	
IDINT	DOUBLE	INTEGER	
AMAX0	INTEGER (>1)	REAL	Choose largest argument
AMAX1	REAL (>1)	REAL	
MAX0	INTEGER (>1)	INTEGER	
MAX1	REAL (>1)	INTEGER	
DMAX1	DOUBLE (>1)	DOUBLE	
AMIN0	INTEGER (>1)	REAL	Choose smallest argument
AMIN1	REAL (>1)	REAL	
MIN0	INTEGER (>1)	INTEGER	
MIN1	REAL (>1)	INTEGER	
DMIN	DOUBLE (>1)	DOUBLE	
FLOAT	INTEGER	REAL	Convert argument mode
IFIX	REAL	INTEGER	
SNGL	DOUBLE	REAL	
REAL	COMPLEX	REAL	
AIMAG	COMPLEX	REAL	
DBLE	REAL	DOUBLE	
CMPLX	REAL (2)	COMPLEX	
SIGN	REAL (2)	REAL	Transfer sign
ISIGN	INTEGER (2)	INTEGER	
DSGN	DOUBLE (2)	DOUBLE	
DIM	REAL (2)	REAL	Positive difference
IDIM	INTEGER (2)	INTEGER	
CONJG	COMPLEX	COMPLEX	Complex conjugate

APPENDIX E
COMPILER ERROR MESSAGES

DDP-116/-516

ERROR MESSAGE	CONDITION
AE	Arithmetic statement function has over 10 arguments
AG	Subroutine or array name not in an argument
AR	Item not an array name
BD	Code generated within a block data subprogram
BL	Block data not first statement
CE	Constant's exponent exceeds 8 bits (over 255)
CG	Compiler or computer error caused a jump to 00000
CH	Improper terminating character (punctuation)
CM	Comma outside parenthesis, not in a DO statement
CN	Improper constant (data initialization)
CR	Illegal common reference
DA	Illegal use of a dummy argument
DD	Dummy item appears in an equivalence or data list
DM	Data and data name mode do not agree
DT	Improper DO termination
EC	Equivalence group not followed by comma or CR (carriage return)
EQ	Expression to left of equals, or multiple equals
EX	Specification statement appears after cleanup
FA	Function has no arguments
FD	Function name not defined by an arithmetic statement
FR	Format statement error
FS	Function/subroutine not the first statement
HF	Hollerith character count equals zero
HS	Hollerith data string extends past end of statement
IC	Impossible common equivalencing
ID	Unrecognizable statement

ERROR MESSAGE	CONDITION
IE	Impossible equivalence grouping
IF	Illegal IF statement type
IN	Integer required at this position
IT	Item not an integer
MM	Mode mixing error
MO	Data pool overflow
MS	Multiply defined statement number
NC	Constant must be present
ND	Wrong number of dimensions
NF	No reference to format statement
NR	Item not a relative variable
NS	Subprogram name not allowed
NT	Logical NOT, not an unary operator
NU	Name already being used
NZ	Non-zero string test failed
OP	More than one operator in a row
PA	Operation must be within parenthesis
PH	No path leading to this statement
PR	Parenthesis missing in a DO statement.
PW	*Preceded by operator other than another*
RL	More than 1 relational operator in a relational example
RN	Reference to a specification statement's number
RT	Return not allowed in main program
SC	Statement number on a continuation card
SP	Statement name misspelled
ST	Illegal statement number format
SU	Subscript incrementer not a constant
TF	"Type" not followed by "Function" or list
TO	Assign statement has word TO missing
UO	Multiple + or — signs, not as unary operators
US	Undefined statement number
VD	Symbolic subscript no dummy in dummy array or symbolic subscript appears on a non-dummy array
VN	Variable name required at this position

ERROR MESSAGE	CONDITION
ADJD	Illegal adjustable dimension
ASOV	Assignment table overflow
ASTO	Word TO incorrect in assign statement
BLKD	Instruction or data generated by block-date subprogram
CICD	Cannot initialize COMMON data
COMM	Illegal common reference
CONS	Illegal constant
CRET	Carriage return within Hollerith string
DDST	Doubly defined statement number
DPFL	Data pool full
DUMM	Illegal dummy appearance
EQCN	Illegal equivalence construction
EQIV	Impossible equivalence
EQMS	Equal (=) sign missing
ERDO	Illegal statement looks like a DO
ERR.	Decimal point missing
ERTN	RETURN not in subprogram
EXS=	Not the first equal sign or illegal (=) sign
FNUM	Numeric value in format statement missing, zero or negative
FOPN	Parenthesis nest in format statement greater than two
FRST	Function or subroutine that is not first statement
FUNV	Either subroutine name used as variable or function not used
FWAR	Function without arguments
IFER	Illegal IF statement
ILBD	Illegal block-data usage
ILEG	Illegal FORTRAN statement
ILSN	Illegal statement number
INDT	Insufficient data for names given
INTG	Noninteger subscript variable

ERROR MESSAGE	CONDITION
IODL	Illegal implied DO loop within I/O statement
IUSE	Incorrect iem usage
LDOP	Improper leading operator
MODE	Mode mixing error
MULT	Multiply defined statement
NARR	Item is not an array
NCBD	Non-common variable in block-data
NCBS	Negative COMMON base
NEST	DOP loop error improper nesting or termination
NINT	Mode is wrong
No (	Subroutine or array name not followed by an open parenthesis or comma missing
NPTH	Format statement without number
OPER	Unacceptable operator or character
OPOS	Operator at illegal position
PATH	Statement without path or unexecutable statement
RLOP	Two relational operators in a row
SBSC	Incorrect number of subscripts
SPEC	Specification statement within program
SPEL	Misspelled FORTRAN statement
STNO	Error in statement number
TMDT	Too much data for names given
TYPE	Illegal name in type statement
UNRF	Unreferenced item
V/SP	Variable that is a subprogram
XARG	More than ten arithmetic statement functions
) (O	Illegal parenthesis in a common statement
) ERR	Right parenthesis error
(ERR	Left parenthesis error
/ ERR	Slash error
, ERR	Comma error
(CR)	Carriage return error

APPENDIX F

PROPOSED USASI FORTRAN IV

The following Proposed American Standard of the FORTRAN language was developed by X3.4.3–FORTRAN Group under the American Standards Association Sectional Committee X3, Computers and Information Processing. The committee was established under the sponsorship of the Business Equipment Manufacturers Association. Here is presented the most recent issue of the proposed standard available at this printing. Any further issues are not expected to alter the technical content.

Inquiries regarding copies of the Proposed Standard should be addressed to the X3 Secretary, BEMA, 235 E. 42nd Street, New York, N.Y.

TABLE OF CONTENTS

PROPOSED USASI FORTRAN IV

1. INTRODUCTION

1.1 Purpose. This standard establishes the form for and the interpretation of programs expressed in the FORTRAN language for the purpose of promoting a high degree of interchangeability of such programs for use on a variety of automatic data processing systems. A processor shall conform to this standard provided it accepts, and interprets as specified, at least those forms and relationships described herein.

Insofar as the interpretation of the form and relationships described are not affected, any statement of requirement could be replaced by a statement expressing that the standard does not provide an interpretation unless the requirement is met. Further, any statement of prohibition could be replaced by a statement expressing that the standard does not provide an interpretation when the prohibition is violated.

1.2 Scope. This standard establishes:

(1) The form of a program written in the FORTRAN language.

(2) The form of writing input data to be processed by such a program operating on automatic data processing systems.

(3) Rules for interpreting the meaning of such a program.

(4) The form of the output data resulting from the use of such a program on automatic data processing systems, provided that the rules of interpretation establish an interpretation.

This standard does not prescribe:

(1) The mechanism by which programs are transformed for use on a data processing system (the combination of this mechanism and data processing system is called a processor).

(2) The method of transcription of such programs or their input or output data to or from a data processing medium.

(3) The manual operations required for set-up and control of the use of such programs on data processing equipment.

(4) The results when the rules for interpretation fail to establish an interpretation of such a program.

(5) The size or complexity of a program that will exceed the capacity of any specific data processing system or the capability of a particular processor.

(6) The range or precision of numerical quantities.

2. BASIC TERMINOLOGY

This section introduces some basic terminology and some concepts. A rigorous treatment of these is given in later sections. Certain assumptions concerning the meaning of grammatical forms and particular words are presented.

A program that can be used as a self-contained computing procedure is called an *executable program* (9.1.6).

An executable program consists of precisely one main program and possibly one or more subprograms (9.1.6).

A *main program* is a set of statements and comments not containing a FUNCTION, SUBROUTINE, or BLOCK DATA statement (9.1.5).

A *subprogram* is similar to a main program but is headed by a BLOCK DATA, FUNCTION, or SUBROUTINE statement. A subprogram headed by a BLOCK DATA statement is called a specification subprogram. A subprogram headed by a FUNCTION or SUBROUTINE statement is called a procedure subprogram (9.1.3, 9.1.4).

The term *program unit* will refer to either a main program or subprogram (9.1.7).

Any program unit except a specification subprogram may reference an *external procedure* (Section 9).

An external procedure that is defined by FORTRAN statements is called a *procedure subprogram*. External procedures also may be defined by other means. An external procedure may be an external function or an external subroutine. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a *function subprogram*. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a *subroutine subprogram* (Sections 8 and 9).

Any program unit consists of *statements* and *comments*. A statement is divided into physical sections called *lines*, the first of which is called an *initial line* and the rest of which are called *continuation lines* (3.2).

There is a type of line called a comment that is not a statement and merely provides information for documentary purposes (3.2).

The statements in FORTRAN fall into two broad classes—executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement (7.1, 7.2).

The syntactic elements of a statement are *names* and *operators*. Names are used to reference objects such as data or procedures. Operators, including the imperative verbs, specify action upon named objects.

One class of name, the *array name*, deserves special mention. An array name must have the size of the identified array defined in an array declarator (7.2.1.1). An array name qualified only by a subscript is used to identify a particular element of the array (5.1.3).

Data names and the arithmetic (or logical) operations may be connected into expressions. Evaluation of such an expression develops a value. This value is derived by performing the specified operations on the named data.

The identifiers used in FORTRAN are names and numbers. Data are named. Procedures are named. Statements are labeled with numbers. Input/output units are numbered (Sections 3, 6, 7).

At various places in this document there are statements with associated lists of entries. In all cases the list is assumed to contain at lease one entry unless an explicit exception is stated. As an example, in the statement

SUBROUTINE $s\ (a_1, a_2, \cdots a_n)$

it is assumed that at least one symbolic name is included in the list within parentheses. A *list* is a set of identifiable elements each of which is separated from its successor by a comma. Further, in a sentence a plural form of a noun will be assumed to also specify the singular form of that noun as a special case when the context of the sentence does not prohibit this interpretation.

The term *reference* is used as a verb with special meaning as defined in Section 5.

3. PROGRAM FORM

Every program unit is constructed of characters grouped into lines and statements.

3.1 The Fortran Character Set. A program unit is written using the following characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	*Name of Character*
	Blank
=	Equals
+	Plus
−	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol

The order in which the characters are listed does not imply a collating sequence.

3.1.1 *Digits.* A digit is one of the ten characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Unless specified otherwise, a string of digits will be interpreted in the decimal base number system when a number system base interpretation is appropriate.

An octal digit is one of the eight characters: 0, 1, 2, 3, 4, 5, 6, 7. These are only used in the STOP (7.1.2.7.1) and PAUSE (7.1.2.7.2) statements.

3.1.2 *Letters.* A letter is one of the twenty-six characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

3.1.3 *Alphanumeric Characters.* An alphanumeric character is a letter or a digit.

3.1.4 *Special Characters.* A special character is one of the eleven characters blank, equals, plus, minus, asterisk, slash, left parenthesis, right parenthesis, comma, decimal point, and currency symbol.

3.1.4.1 *Blank Character.* With the exception of the uses specified (3.2.2, 3.2.3, 3.2.4, 4.2.6, 5.1.1.6, 7.2.3.6, and 7.2.3.8), a blank character has no meaning and may be used freely to improve the appearance of the program subject to the restriction on continuation lines in 3.3.

3.2 Lines. A line is a string of 72 characters. All characters must be from the Fortran character set except as described in 5.1.1.6 and 7.2.3.8.

The character positions in a line are called columns and are consecutively numbered 1, 2, 3, ⋯, 72. The number indicates the sequential position of a character in the line starting at the left and proceeding to the right.

3.2.1 *Comment Line.* The letter C in column 1 of a line designates that line as a comment line. A comment line must be immediately followed by an initial line, another comment line, or an end line.

A comment line does not affect the program in any way and is available as a convenience for the programmer.

3.2.2 *End Line.* An end line is a line with the character blank in columns 1 through 6, the characters E, N, and D, once each and in that order, in columns 7 through 72, preceded by, interspersed with, or followed by the character blank. The end line indicates to the processor, the end of the written description of a program unit (9.1.7). Every program unit must physically terminate with an end line.

3.2.3 *Initial Line.* An initial line is a line that is neither a comment line nor an end line and that contains the digit 0 or the character blank in column 6. Columns 1 through 5 contain the statement label or each contains the character blank.

3.2.4 *Continuation Line.* A continuation line is a line that contains any character other than the digit 0 or the character blank in column 6, and does not contain the character C in column 1.

A continuation line may only follow an initial line or another continuation line.

3.3 Statements. A statement consists of an initial line optionally followed by up to nineteen ordered continuation lines. The statement is written in columns 7 through 72 of the lines. The order of the characters in the statement is columns 7 through 72 of the initial line followed, as applicable, by columns 7 through 72 of the first continuation line, columns 7 through 72 of the next continuation line, etc.

3.4 Statement Label. Optionally, a statement may be labeled so that it may be referred to in other statements. A statement label consists of from one to five digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label may not be given to more than one statement in a program unit. Leading zeros are not significant in differentiating statement labels.

3.5 Symbolic Names. A symbolic name consists of from one to six alphanumeric characters, the first of which must be alphabetic. See 10.1 through 10.1.10 for a discussion of classification of symbolic names and restrictions on their use.

3.6 Ordering of Characters. An ordering of characters is assumed within a program unit. Thus, any meaningful collection of characters that constitutes names, lines, and statements exists as a totally ordered set. This ordering is imposed by the character position rule of 3.2 (which orders characters within lines) and the order in which lines are presented for processing.

4. DATA TYPES

Six different types of data are defined. These are integer, real, double precision, complex, logical, and Hollerith. Each type has a different mathematical significance and may have different internal representation. Thus the data type has a significance in the interpretation of the associated operations with which a datum is involved. The data type of a function defines the type of the datum it supplies to the expression in which it appears.

4.1 Data Type Association. The name employed to identify a datum or function carries the data type association. The form of the string representing a constant defines both the value and the data type.

A symbolic name representing a function, variable, or array must have only a single data type association for each program unit. Once associated with a particular data type, a specific name implies that type for any differing usage of that symbolic name that requires a data type association throughout the program unit in which it is defined.

Data type may be established for a symbolic name by declaration in a type-statement (7.2.1.6) for the integer, real, double precision, complex, and logical types. This specific declaration overrides the implied association available for integer and real (5.3).

There exists no mechanism to associate a symbolic name with the Hollerith data type. Thus data of this type, other than constants, are identified under the guise of a name of one of the other types.

4.2 Data Type Properties. The mathematical and the representation properties for each of the data types are defined in the following sections. For real, double precision, and integer data, the value zero is considered neither positive nor negative.

4.2.1 *Integer Type.* An integer datum is always an exact representation of an integer value. It may assume positive, negative, and zero values. It may only assume integral values.

4.2.2 *Real Type.* A real datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values.

4.2.3 *Double Precision Type.* A double precision datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values. The degree of approximation, though undefined, must be greater than that of type real.

4.2.4 *Complex Type.* A complex datum is a processor approximation to the value of a complex number. The representation of the approximation is in the form of an ordered pair of real data. The first of the pair represents the real part and the second, the imaginary part. Each part has, accordingly, the same degree of approximation as for a real datum.

4.2.5 *Logical Type.* A logical datum may assume only the truth values of true or false.

4.2.6 *Hollerith Type.* A Hollerith datum is a string of characters. This string may consist of any characters capable of representation in the processor. The blank character is a valid and significant character in a Hollerith datum.

5. DATA AND PROCEDURE IDENTIFICATION

Names are employed to reference or otherwise identify data and procedures.

The term *reference* is used to indicate an identification of a datum implying that the current value of the datum will be made available during the execution of the statement containing the reference. If the datum is identified but not necessarily made available, the datum is said to be *named.* One case of special interest in which the datum is named is that of assigning a value to a datum, thus defining or redefining the datum.

The term, reference, is used to indicate an identification of a procedure implying that the actions specified by the procedure will be made available.

A complete and rigorous discussion of reference and definition, including redefinition, is contained in Section 10.

5.1 Data and Procedure Names. A data name identifies a constant, a variable, an array or array element, or a block (7.2.1.3). A procedure name identifies a function or a subroutine.

5.1.1 *Constants.* A constant is a datum that is always defined during execution and may not be redefined. Rules for writing constants are given for each data type.

An integer, real, or double precision constant is said to be signed when it is written immediately following a plus or minus. Also, for these types, an optionally signed constant is either a constant or a signed constant.

5.1.1.1 *Integer constant.* An integer constant is written as a nonempty string of digits. The constant is the digit string interpreted as a decimal numeral.

5.1.1.2 *Real Constant.* A basic real constant is written as an integer part, a decimal point, and a decimal fraction part in that order. Both the integer part and the decimal part are strings of digits; either one of these strings may be empty but not both. The constant is an approximation to the digit string interpreted as a decimal numeral.

A decimal exponent is written as the letter, E, followed by an optionally signed integer constant. A decimal exponent is a multiplier (applied to the constant written immediately preceding it) that is an approximation to the exponential form ten raised to the power indicated by the integer written following the E.

A real constant is indicated by writing a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent.

5.1.1.3 *Double Precision Constant.* A double precision exponent is written and interpreted identically to a decimal exponent except that the letter, D, is used instead of the letter, E.

A double precision constant is indicated by writing a basic real constant followed by a double precision exponent or an integer constant followed by a double precision exponent.

5.1.1.4 *Complex Constant.* A complex constant is written as an ordered pair of optionally signed real constants, separated by a comma, and enclosed within parentheses. The datum is an approximation to the complex number represented by the pair.

5.1.1.5 *Logical Constant.* The logical constants, true and false, are written .TRUE. and .FALSE. respectively.

5.1.1.6 *Hollerith Constant.* A Hollerith constant is written as an integer constant (whose value *n* is greater than zero) followed by the letter H, followed by exactly *n* characters which comprise the Hollerith datum proper. Any *n* characters capable of representation by the processor may follow the H. The character blank is significant in the Hollerith datum string. This type of constant may be written only in the argument list of a CALL statement and in the data initialization statement.

5.1.2 *Variable.* A variable is a datum that is identified by a symbolic name (3.5). Such a datum may be referenced and defined.

5.1.3 *Array.* An array is an ordered set of data of one, two, or three dimensions. An array is identified by a symbolic name. Identification of the entire ordered set is achieved via use of the array name.

5.1.3.1 *Array Element.* An array element is one of the members of the set of data of an array. An array element is identified by immediately following the array name with a qualifier, called a subscript, which points to the particular element of the array.

An array element may be referenced and defined.

5.1.3.2 *Subscript.* A subscript is written as a parenthesized list of subscript expressions. Each subscript expression is separated by a comma from its successor, if there is a successor. The number of subscript expressions must correspond to the declared dimensionality (7.2.1.1), except in an EQUIVALENCE statement (7.2.1.4). Following evaluation of all of the subscript expressions, the array element successor function (7.2.1.1) determines the identified array element.

5.1.3.3 *Subscript Expressions.* A subscript expression is written as one of the following constructs:

$$c^*v + k$$
$$c^*v - k$$
$$c^*v$$
$$v + k$$
$$v - k$$
$$v$$
$$k$$

where c and k are integer constants and v is an integer variable reference. See Section 6 for a discussion of evaluation of expressions and 10.2.8 and 10.3 for requirements that apply to the use of a variable in a subscript.

5.1.4 *Procedures.* A procedure (Section 8) is identified by a symbolic name. A procedure is a statement function, an intrinsic function, a basic external function, an external function, or an external subroutine. Statement functions, intrinsic functions, basic external functions, and external functions are referred to as functions or function procedures; external subroutines as subroutines or subroutine procedures.

A function supplies a result to be used at the point of reference; a subroutine does not. Functions are referenced in a manner different from subroutines.

5.2 Function Reference. A function reference consists of the function name followed by an actual argument list enclosed in parentheses. If the list contains more than one argument, the arguments are separated by commas. The allowable forms of function arguments are given in Section 8.

See 10.2.1 for a discussion of requirements that apply to function references.

5.3 Type Rules for Data and Procedure Identifiers. The type of a constant is implicit in its name.

There is no type associated with a symbolic name that identifies a subroutine or a block.

A symbolic name that identifies a variable, an array, or a statement function may have its type specified in a type-statement. In the absence of an explicit declaration, the type is implied by the first character of the name: I, J, K, L, M, and N imply type integer; any other letter implies type real.

A symbolic name that identifies an intrinsic function or a basic external function when it is used to identify this designated procedure, has a type associated with it as specified in Tables 3 and 4.

In the program unit in which an external function is referenced, its type definition is defined in the same manner as for a variable and an array. For a function subprogram, type is specified either implicitly by its name or explicitly in the FUNCTION statement.

The same type is associated with an array element as is associated with the array name.

5.4 Dummy Arguments. A dummy argument of an external procedure identifies a variable, array, subroutine, or external function.

When the use of an external function name is specified, the use of a dummy argument is permissible if an external function name will be associated with that dummy argument. (Section 8.)

When the use of an external subroutine name is specified, the use of a dummy argument is permissible if an external subroutine name will be associated with that dummy argument.

When the use of a variable or array element reference is specified, the use of a dummy argument is permissible if a value of the same type will be made available through argument association.

Unless specified otherwise, when the use of a variable, array, or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

The process of argument association is discussed in Sections 8 and 10.

6. EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic, relational, and logical expressions. A relational expression appears only within the context of logical expressions. An expression is formed from elements and operators. See 10.3 for a discussion of requirements that apply to the use of certain entities in expressions.

6.1 Arithmetic Expressions. An arithmetic expression is formed with arithmetic operators and arithmetic elements. Both the expression and its constituent elements identify values of one of the types integer, real, double precision, or complex. The arithmetic operators are:

Operator	*Representing*
+	Addition, positive value (zero + element)
−	Subtraction, negative value (zero − element)
*	Multiplication
/	Division
**	Exponentiation

The arithmetic elements are primary, factor, term, signed term, simple arithmetic expression, and arithmetic expression.

A primary is an arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

A factor is a primary or a construct of the form

*primary**primary*

A term is a factor or a construct of one of the forms

term/factor

or

*term*term*

A signed term is a term immediately preceded by + or −.

A simple arithmetic expression is a term or two simple arithmetic expressions separated by a + or −.

An arithmetic expression is a simple arithmetic expression or a signed term or either of the preceding forms immediately followed by a + or − immediately followed by a simple arithmetic expression.

A primary of any type may be exponentiated by an integer primary, and the resultant factor is of the same type as that of the element being exponentiated. A real or double precision primary may be exponentiated by a real or double precision primary, and the resultant factor is of type real if both primaries are of type real and otherwise of type double precision. These are the only cases for which use of the exponentiation operator is defined.

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type. Further, an admissible real element may be combined with an admissible double precision or complex element; the resultant element is of type double precision or complex, respectively.

6.2 Relational Expressions. A relational expression consists of two arithmetic expressions separated by a relational operator and will have the value true or false as the relation is true or false, respectively. One arithmetic expression may be of type real or double precision and the other of type real or double precision, or both arithmetic expressions may be of type integer. If a real expression and a double precision expression appear in a relational expression, the effect is the same as a similar relational expression. This similar expression contains a double precision zero as the right hand arithmetic expression and the difference of the two original expressions (in their original order) as the left. The relational operator is unchanged. The relational operators are:

Operator	*Representing*
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

6.3 Logical Expressions. A logical expression is formed with logical operators and logical elements and has the value true or false. The logical operators are:

Operator	*Representing*
.OR.	Logical disjunction
.AND.	Logical conjunction
.NOT.	Logical negation

The logical elements are logical primary, logical factor, logical term, and logical expression.

A logical primary is a logical expression enclosed in parentheses, a relational expression, a logical constant, a logical variable reference, a logical array element reference, or a logical function reference.

A logical factor is a logical primary or .NOT. followed by a logical primary.

A logical term is a logical factor or a construct of the form:

logical term .AND. logical term

A logical expression is a logical term or a construct of the form:

logical expression .OR. logical expression

6.4 Evaluation of Expressions. A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence (except as modified in the following paragraph).

When two elements are combined by an operator, the order of evaluation of the elements is optional. If mathematical use of operators is associative, commutative, or both, full use of these facts may be made to revise orders of combination, provided only that integrity of parenthesized expressions is not violated. The results of different permissible orders of combination even though mathematically identical need not be computationally identical. The value of an integer factor or term is the nearest integer whose magnitude does not exceed the magnitude of the mathematical value represented by that factor or term. The associative and commutative laws do not apply in the evaluation of integer terms containing division, hence the evaluation of such terms must effectively proceed from left to right.

Any use of an array element name requires the evaluation of its subscript. The evaluation of functions appearing in an expression may not validly alter the value of any other element within the expressions, assignment statement, or CALL statement in which the function reference appears. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript.

No factor may be evaluated that requires a negative valued primary to be raised to a real or double precision exponent. No factor may be evaluated that requires raising a zero valued primary to a zero valued exponent.

No element may be evaluated whose value is not mathematically defined.

7. STATEMENTS

A statement may be classified as executable or nonexecutable. Executable statements specify actions; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and classification of program units.

7.1 Executable Statements. There are three types of executable statements:

(1) Assignment statements.
(2) Control statements.
(3) Input output statements.

7.1.1 *Assignment Statements.* There are three types of assignment statements:

(1) Arithmetic assignment statement.
(2) Logical assignment statement.
(3) GO TO assignment statement.

7.1.1.1 *Arithmetic Assignment Statement.* An arithmetic assignment statement is of the form:

$$v = e$$

where v is a variable name or array element name of type other than logical and e is an arithmetic expression. Execution of this statement causes the evaluation of the expression e and the altering of v according to Table 1.

7.1.1.2 *Logical Assignment Statement.* A logical assignment statement is of the form

$$v = e$$

where v is a logical variable name or a logical array element name and e is a logical expression. Execution of this statement causes the logical expression to be evaluated and its value to be assigned to the logical entity.

7.1.1.3 GO TO *Assignment Statement.* A GO TO assignment statement is of the form:

ASSIGN k TO i

where k is a statement label and i is an integer variable name. After execution of such a statement, subsequent execution of any assigned GO TO statement (Section 7.1.2.1.2) using that integer variable will cause the statement identified by the assigned statement label to be executed next, provided there has been no intervening redefinition (9.2) of the variable. The statement label must refer to an executable statement in the same program unit in which the ASSIGN statement appears.

Once having been mentioned in an ASSIGN statement, an integer variable may not be referenced in any statement other than an assigned GO TO statement until it has been redefined (Section 10.2.3).

TABLE 1. RULES FOR ASSIGNMENT OF e TO v

If v Type Is	*And e Type Is*	*The Assignment Rule Is**
Integer	Integer	Assign
Integer	Real	Fix & Assign
Integer	Double Precision	Fix & Assign
Integer	Complex	P
Real	Integer	Float & Assign
Real	Real	Assign
Real	Double Precision	DP Evaluate & Real Assign
Real	Complex	P
Double Precision	Integer	DP Float & Assign
Double Precision	Real	DP Evaluate & Assign
Double Precision	Double Precision	Assign
Double Precision	Complex	P
Complex	Integer	P
Complex	Real	P
Complex	Double Precision	P
Complex	Complex	Assign

*NOTES.

(1) P means prohibited combination.

(2) Assign means transmit the resulting value, without change, to the entity.

(3) Real Assign means transmit to the entity as much precision of the most significant part of the resulting value as a real datum can contain.

(4) DP Evaluate means evaluate the expression according to the rules of 6.1 (or any more precise rules) then DP Float.

(5) Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.

(6) Float means transform the value to the form of a real datum.

(7) DP Float means transform the value to the form of a double precision datum, retaining in the process as much of the precision of the value as a double precision datum can contain.

7.1.2 *Control Statements.* There are eight types of control statements:

(1) GO TO statements.
(2) arithmetic IF statement.
(3) logical IF statement.
(4) CALL statement.
(5) RETURN statement.
(6) CONTINUE statement.
(7) program control statements.
(8) DO statement.

The statement labels used in a control statement must be associated with executable statements within the same program unit in which the control statement appears.

7.1.2.1 GO TO *Statements.* There are three types of GO TO statements:

(1) Unconditional GO TO statement.
(2) Assigned GO TO statement.
(3) Computed GO TO statement.

7.1.2.1.1 *Unconditional* GO TO *Statement.* An unconditional GO TO statement is of the form:

GO TO k

where k is a statement label.

Execution of this statement causes the statement identified by the statement label to be executed next.

7.1.2.1.2 *Assigned* GO TO *Statement.* An assigned GO TO statement is of the form:

GO TO i, $(k_1, k_2, \cdots, k_n)$

where i is an integer variable reference, and the k's are statement labels.

At the time of execution of an assigned GO TO statement, the current value of i must have been assigned by the previous execution of an ASSIGN statement to be one of the statement labels in the parenthesized list, and such an execution causes the statement identified by that statement label to be executed next.

7.1.2.1.3 *Computed* GO TO *Statement.* A computed GO TO statement is of the form:

GO TO $(k_1, k_2, \cdots, k_n)$, i

where the k's are statement labels and i is an integer variable reference. See 10.2.8 and 10.3 for a discussion of requirements that apply to the use of a variable in a computed GO TO statement.

Execution of this statement causes the statement identified by the statement label k_j to be executed next, where j is the value of i at the time of the execution. This statement is defined only for values such that $1 \leqq j \leqq n$.

7.1.2.2 *Arithmetic* IF *Statement.* An arithmetic IF statement is of the form:

IF (e) k_1, k_2, k_3

where e is any arithmetic expression of type integer, real, or double precision, and the k's are statement labels.

The arithmetic IF is a three-way branch. Execution of this statement causes evaluation of the expression e following which the statement identified by the statement label k_1, k_2, or k_3 is executed next as the value of e is less than zero, zero, or greater than zero, respectively.

7.1.2.3 *Logical* IF *Statement.* A logical IF statement is of the form:

IF (e) S

where e is a logical expression and S is any executable statement except a DO statement or another logical IF statement. Upon execution of this statement, the logical expression e is evaluated. If the value of e is false, statement S is executed as though it were a CONTINUE statement. If the value of e is true, statement S is executed.

7.1.2.4 CALL *Statement.* A CALL statement is of one of the forms:

CALL s $(a_1, a_2, \cdots, a_n)$
or
CALL s

where s is the name of a subroutine and the a's are actual arguments (8.4.2).

The inception of execution of a CALL statement references the designated subroutine. Return of control from the designated subroutine completes execution of the CALL statement.

7.1.2.5 RETURN *Statement.* A RETURN statement is of the form:

RETURN

A RETURN statement marks the logical end of a procedure subprogram and, thus, may only appear in a procedure subprogram.

Execution of this statement when it appears in a subroutine subprogram causes return of control to the current calling program unit.

Execution of this statement when it appears in a function subprogram causes return of control to the current calling program unit. At this time the value of the function (8.3.1) is made available.

7.1.2.6 CONTINUE *Statement.* A CONTINUE statement is of the form:

CONTINUE

Execution of this statement causes continuation of normal execution sequence.

7.1.2.7 *Program Control Statements.* There are two types of program control statements:

(1) STOP statement.

(2) PAUSE statement.

7.1.2.7.1 STOP *Statement.* A STOP statement is of one of the forms:

STOP n

or

STOP

where n is an octal digit string of length from one to five.

Execution of this statement causes termination of execution of the executable program.

7.1.2.7.2 PAUSE *Statement.* A PAUSE statement is of one of the forms:

PAUSE n

or

PAUSE

where n is an octal digit string of length from one to five.

The inception of execution of this statement causes a cessation of execution of this executable program. Execution must be resumable. At the time of cessation of execution the octal digit string is accessible. The decision to resume execution is not under control of the program, but if execution is resumed without otherwise changing the state of the processor, the completion of the PAUSE statement causes continuation of normal execution sequence.

7.1.2.8 DO *Statement.* A DO statement is of one of the forms:

DO n i = m_1, m_2, m_3

or

DO n i = m_1, m_2

where:

(1) n is the statement label of an executable statement. This statement, called the terminal statement of the associated DO, must physically follow and be in the same program unit as that DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or DO statement, nor a logical IF containing any of these forms.

(2) i is an integer variable name; this variable is called the control variable.

(3) m_1, called the initial parameter; m_2, called the terminal parameter; and m_3, called the incrementation parameter, are each either an integer constant or integer variable reference. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the DO statement, m_1, m_2, and m_3 must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO. A special situation occurs when the range of a DO contains another DO statement. In this case, the range of the contained DO must be a subset of the range of the containing DO.

A completely nested nest is a set of DO statements and their ranges, and any DO statements contained within their ranges, such that the first occurring terminal statement of any of those DO statements physically follows the last occurring DO statement and the first occurring DO statement of the set is not in the range of any DO statement.

A DO statement is used to define a loop. The action succeeding execution of a DO statement is described by the following five steps:

1. The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

2. The range of the DO is executed.

3. If control reaches the terminal statement, and after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

4. If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal parameter, the action as described starting at step 2 is repeated with the understanding that the range in question is that of the DO, the control variable of which was most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter, the DO is said to have been satisfied and the control variable becomes undefined.

5. At this point, if there were one or more other DO statements referring to the terminal statement in question, the control variable of the next most recently executed DO statement is incremented by the value represented by its associated incrementation parameter and the action as described in step 4 is repeated until all DO statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed. In the remainder of this section (7.1.2.8) a logical IF statement containing a GO TO or arithmetic IF statement form is regarded as a GO TO or arithmetic IF statement respectively.

Upon exiting from the range of a DO by execution of a GO TO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the foregoing.

A DO is said to have an extended range if both of the following conditions apply:

(1) There exists a GO TO statement or arithmetic IF statement within the range of the innermost DO of a completely nested nest that can cause control to pass out of that nest.

(2) There exists a GO TO statement or arithmetic IF statement not within the nest that, in the collection of all possible sequences of execution in the particular program unit could be executed after a statement of the type described in (1), and the execution of which could cause control to return into the range of the innermost DO of the completely nested nest.

If both of these conditions apply, the extended range is defined to be the set of all executable statements that may be executed between all pairs of control statements, the first of which satisfies the condition of (1) and the second of (2). The first of the pair is not included in the extended range; the second is. A GO TO statement or an arithmetic IF statement may not cause control to pass into the range of a DO unless it is being executed as part of the extended range of that particular DO. Further, the extended range of a DO may not contain a DO of the same program unit that has an extended range. When a procedure reference occurs in the range of a DO the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable, initial parameter, terminal parameter, and incrementation parameter of a DO may not be redefined during the execution of the range or extended range of that DO.

If a statement is the terminal statement of more than one DO statement, the statement label of that terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

7.1.3 *Input/Output Statements.* There are two types of input/output statements:

(1) READ and WRITE statements.

(2) Auxiliary Input/Output statements.

The first type consists of the statements that cause transfer of records of sequential files to and from internal storage, respectively. The second type consists of the BACKSPACE and REWIND statements that provide for positioning of such an external file, and ENDFILE, which provides for demarcation of such an external file.

In the following descriptions, *u* and *f* identify input/output units and format specifications, respectively. An input/output unit is identified by an integer value and *u* may be either an integer constant or an integer variable reference whose value then identifies the unit. The format specification is described in Section 7.2.3. Either the statement label of a FORMAT statement or an array name may be represented by *f*. If a statement label, the identified statement must appear in the same program unit as the input/output statement. If an array name, it must conform to the specifications in 7.2.3.10.

A particular unit has a single sequential file associated with it. The most general case of such a unit has the following properties:

(1) If the unit contains one or more records, those records exist as a totally ordered set.

(2) There exists a unique position of the unit called its initial point. If a unit contains no records, that unit is positioned at its initial point. If the unit is at its initial point and contains records, the first record of the unit is defined as the next record.

(3) If a unit is not positioned at its initial point, there exists a unique preceding record associated with that position. The least of any records in the ordering described by (1) following this preceding record is defined as the next record of that position.

(4) Upon completion of execution of a WRITE or ENDFILE statement, there exist no records following the records created by that statement.

(5) When the next record is transmitted, the position of the unit is changed so that this next record becomes the preceding record.

If a unit does not provide for some of the properties given in the foregoing, certain statements that will be defined may not refer to that unit. The use of such a statement is not defined for that unit.

7.1.3.1 READ *and* WRITE *Statements.* The READ and WRITE statements specify transfer of information. Each such statement may include a list of the names of variables, arrays, and array elements. The named elements are assigned values on input and have their values transferred on output.

Records may be formatted or unformatted. A formatted record consists of a string of the characters that are permissible in Hollerith constants (5.1.1.6). The transfer of such a record requires that a format specification be referenced to supply the necessary positioning and conversion specifications (7.2.3). The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format specification (7.2.3.4). An unformatted record consists of a string of values. When an unformatted or formatted READ statement is executed, the required records on the identified unit must be, respectively, unformatted or formatted records.

7.1.3.1.1 *Input/Output Lists.* The input list specifies the names of the variables and array elements to which values are assigned on input. The output list specifies the references to variables and array elements whose values are transmitted. The input and output lists are of the same form.

Lists are formed in the following manner. A simple list is a variable name, an array element name, or an array name, or two simple lists separated by a comma.

A list is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by a comma.

A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses.

A DO-implied specification is of one of the forms:

$$i = m_1, m_2, m_3$$

or

$$i = m_1, m_2$$

The elements i, m_1, m_2, and m_3 are as defined for the DO statement (7.1.2.8). The range of DO-implied specification is the list of the DO-implied list and, for input lists, i, m_1, m_2, and m_3 may appear, within that range, only in subscripts.

A variable name or array element name specifies itself. An array name specifies all of the array element names defined by the array declarator, and they are specified in the order given by the array element successor function (7.2.1.1.1).

The elements of a list are specified in the order of their occurrence from left to right. The elements of a list in a DO-implied list are specified for each cycle of the implied DO.

7.1.3.1.2 *Formatted* READ. A formatted READ statement is of one of the forms:

READ (*u*, *f*) *k*

or

READ (*u*, *f*)

where *k* is a list.

Execution of this statement causes the input of the next records from the unit identified by *u*. The information is scanned and converted as specified by the format specification identified by *f*. The resulting values are assigned to the elements specified by the list. See however 7.2.3.4.

7.1.3.1.3 *Formatted* WRITE. A formatted WRITE statement is of one of the forms:

WRITE (*u*, *f*) *k*

or

WRITE (*u*, *f*)

where *k* is a list.

Execution of this statement creates the next records on the unit identified by *u*. The list specifies a sequence of values. These are converted and positioned as specified by the format specification identified by *f*. See however 7.2.3.4.

7.1.3.1.4 *Unformatted* READ. An unformatted READ statement is of one of the forms:

READ (*u*) *k*

or

READ (*u*)

where *k* is a list.

Execution of this statement causes the input of the next record from the unit identified by u, and, if there is a list, these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

7.1.3.1.5 *Unformatted* WRITE. An unformatted WRITE statement is of the form:

WRITE (u) k

where k is a list.

Execution of this statement creates the next record on the unit identified by u of the sequence of values specified by the list.

7.1.3.2 *Auxiliary Input/Output Statements.* There are three types of auxiliary input/output statements:

(1) REWIND statement.
(2) BACKSPACE statement.
(3) ENDFILE statement.

7.1.3.2.1 REWIND *Statement.* A REWIND statement is of the form:

REWIND u

Execution of this statement causes the unit identified by u to be positioned at its initial point.

7.1.3.2.2 BACKSPACE *Statement.* A BACKSPACE statement is of the form:

BACKSPACE u

If the unit identified by u is positioned at its initial point, execution of this statement has no effect. Otherwise, the execution of this statement results in the positioning of the unit identified by u so that what had been the preceding record prior to that execution becomes the next record.

7.1.3.2.3 ENDFILE *Statement.* An ENDFILE statement is of the form:

ENDFILE u

Execution of this statement causes the recording of an endfile record on the unit identified by u. The endfile record is an unique record signifying a demarcation of a sequential file. Action is undefined when an endfile record is encountered during execution of a READ statement.

7.1.3.3 *Printing of Formatted Record.* When formatted records are prepared for printing, the first character of the record is not printed.

The first character of such a record determines vertical spacing as follows:

Character	*Vertical Spacing Before Printing*
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

7.2 Nonexecutable Statements. There are five types of nonexecutable statements:

(1) Specification statements.
(2) Data initialization statement.
(3) FORMAT statement.
(4) Function defining statements.
(5) Subprogram statements.

See 10.1.2 for a discussion of restrictions on appearances of symbolic names in such statements.

The function defining statements and subprogram statements are discussed in Section 8.

7.2.1 *Specification Statements.* There are five types of specification statements:

(1) DIMENSION statement.
(2) COMMON statement.
(3) EQUIVALENCE statement.
(4) EXTERNAL statement.
(5) Type-statements.

7.2.1.1 *Array-Declarator.* An array declarator specifies an array used in a program unit.

The array declarator indicates the symbolic name, the number of dimensions (one, two, or three), and the size of each of the dimensions. The array declarator statement may be a type-statement, DIMENSION, or COMMON statement.

An array declarator has the form:

v (i)

where:

(1) v, called the declarator name, is a symbolic name,

(2) (i), called the declarator subscript, is composed of 1, 2, or 3 expressions, each of which may be an integer constant or an integer variable name. Each expression is separated by a comma from its successor if there are more than one of them. In the case where i contains no integer variable, i is called the constant declarator subscript.

The appearance of a declarator subscript in a declarator statement serves to inform the processor that the declarator name is an array name. The number of subscript expressions specified for the array indicates its dimensionality. The magnitude of the values given for the subscript expressions indicates the maximum value that the subscript may attain in any array element name.

No array element name may contain a subscript that, during execution of the executable program, assumes a value less than one or larger than the maximum length specified in the array declarator.

7.2.1.1.1 *Array Element Successor Function and Value of a Subscript.* For a given dimensionality, subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain is indicated in Table 2. A subscript expression must be greater than zero.

The value of the array element successor function is obtained by adding one to the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and has no succcssor element.

TABLE 2. Value of a Subscript

Dimensionality	*Subscript Declarator*	*Subscript*	*Subscript Value*	*Maximum Subscript Value*
1	(A)	(a)	a	$A \cdot$
2	(A, B)	(a, b)	$a + A \cdot (b - 1)$	$A \cdot B$
3	(A, B, C)	(a, b, c)	$a + A \cdot (b - 1) + A \cdot B \cdot (c - 1)$	$A \cdot B \cdot C$

Notes. (1) a, b, and c are subscript expressions.
(2) A, B, and C are dimensions.

7.2.1.1.2 *Adjustable Dimension.* If any of the entires in a declarator subscript is an integer variable name, the array is called an adjustable array, and the variable names are called adjustable dimensions. Such an array may only appear in a procedure subprogram. The dummy argument list of the subprograms must contain the array name and the integer variable names that represent the adjustable dimensions. The values of the actual arguments that represent array dimensions in the argument list of the reference must be defined (10.2) prior to calling the subprogram and may not be redefined or undefined during execution of the subprogram. The maximum size of the actual array may not be exceeded. For every array appearing in an executable program (9.1.6), there must be at least one constant array declarator associated through subprogram.

In a subprogram, a symbolic name that appears in a COMMON statement may not identify an adjustable array.

7.2.1.2 DIMENSION *Statement.* A DIMENSION statement is of the form:

DIMENSION $v_1(i_1)$, $v_2(i_2)$, $\cdots$, $v_n(i_n)$

where each $v(i)$ is an array declarator.

7.2.1.3 COMMON *Statement.* A COMMON statement is of the form:

COMMON / x_1 / a_1 / $\cdots$ / x_n / a_n

where each a is a nonempty list of variable names, array names, or array declarators (no dummy arguments are permitted) and each x is a symbolic name or is empty. If x_1 is empty, the first two slashes are optional. Each x is a block name, a name that bears no relationship to any variable or array having the same name. This holds true for any such variable or array in the same or any other program unit. See 10.1.1 for a discussion of restrictions on uses of block names.

In any given COMMON statement, the entities occurring between block name x and the next block name (or the end of the statement if no block name follows) are declared to be in common block x. All entities from the beginning of the statement until the appearance of a block name, or all entities in the statement if no block name appears, are declared to be in blank or unlabeled common. Alternatively, the appearance of two slashes with no block name between them declares the entities that follow to be in blank common.

A given common block name may occur more than once in a COMMON statement or in a program unit. The processor will string together in a given common block all entities so assigned in the order of their appearance (10.1.2). The first element of an array will follow the immediately preceding entity, if one exists, and the last element of an array will immediately precede the next entity, if one exists.

The size of a common block in a program unit is the sum of the storage required for the elements introduced through COMMON and EQUIVALENCE statements. The sizes of labeled common blocks with the same label in the program units that comprise an executable program must be the same. The sizes of blank common in the various program units that are to be executed together need not be the same. Size is measured in terms of storage units (7.2.1.3.1).

7.2.1.3.1 *Correspondence of Common Blocks.* If all of the program units of an executable program that contain any definition of a common block of a particular name define that block such that:

(1) There is identity in type for all entities defined in the corresponding position from the beginning of that block,

(2) If the block is labeled and the same number of entities is defined for the block.

Then the values in the corresponding positions (counted by the number of preceding storage units) are the same quantity in the executable program.

A double precision or a complex entity is counted as two logically consecutive storage units; a logical, real, or integer entity, as one storage unit.

Then for common blocks with the same number of storage units or blank common:

(1) In all program units which have defined the identical type to a given position (counted by the number of preceding storage units) references to that position refer to the same quantity.

(2) A correct reference is made to a particular position assuming a given type if the most recent value assignment to that position was of the same type.

7.2.1.4 EQUIVALENCE *Statement.* An EQUIVALENCE statement is of the form:

EQUIVALENCE (k_1), (k_2), $\cdots$, (k_n)

in which each k is a list of the form:

$a_1, a_2, \cdots , a_m.$

Each a is either a variable name or an array element name (not a dummy argument), the subscript of which contains only constants, and m is greater than or equal to two. The number of subscript expressions of an array element name must correspond in number to the dimensionality of the array declarator or must he one (the array element successor function defines a relation by which an array can be made equivalent to a one dimensional array of the same length).

The EQUIVALENCE statement is used to permit the sharing of storage by two or more entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor. The EQUIVALENCE statement should not be used to equate mathematically two or more entities. If a two storage unit entity is equivalenced to a one storage unit entity, the latter will share space with the first storage unit of the former.

The assignment of storage to variables and arrays declared directly in a COMMON statement is determined solely by consideration of their type and the COMMON and array declarator statements. Entities so declared are always assigned unique storage, contiguous in the order declared in the COMMON statement.

The effect of an EQUIVALENCE statement upon common assignment may be the lengthening of a common block; the only such lengthening permitted is that which extends a common block beyond the last assignment for that block made directly by a COMMON statement.

When two variables or array elements share storage because of the effects of EQUIVALENCE statements, the symbolic names of the variables or arrays in question may not both appear in COMMON statements in the same program unit.

Information contained in 7.2.1.1.1, 7.2.1.3.1, and the present section suffices to describe the possibilities of additional cases of sharing of storage between array elements and entities of common blocks. It is incorrect to cause either directly or indirectly a single storage unit to contain more than one element of the same array.

7.2.1.5 EXTERNAL *Statement.* An EXTERNAL statement is of the form:

EXTERNAL v_1, v_2, $\cdots$, v_n

where each v is an external procedure name.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name. If an external procedure name is used as an argument to another external procedure, it must appear in an EXTERNAL statement in the program unit in which it is so used.

7.2.1.6 *Type-statements.* A type-statement is of the form:

t v_1, v_2, $\cdots$, v_n

where t is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, and each v is a variable name, an array name, a function name, or an array declarator.

A type-statement is used to override or confirm the implicit typing, to declare entities to be of type double precision, complex, or logical, and may supply dimension information.

The appearance of a symbolic name in a type-statement serves to inform the processor that it is of the specified data type for all appearances in the program unit. See, however, the restriction in 8.3.1 second paragraph.

7.2.2 *Data Initialization Statement.* A data initialization statement is of the form:

DATA k_1 / d_1 / , k_2 / d_2 / , $\cdots$, k_n / d_n /

where:

(1) Each k is a list containing names of variables and array elements,

(2) Each d is a list of constants and optionally signed constants, any of which may be preceded by j*,

(3) j is an integer constant.

If a list contains more than one entry, the entries are separated by commas.

Dummy arguments may not appear in the list k. Any subscript expression must be an integer constant.

When the form j* appears before a constant it indicates that the constant is to be specified j times. A Hollerith constant may appear in the list d.

A data initialization statement is used to define initial values of variables or array elements. There must be a one-to-one correspondence between the list-specified items and the constants. By this correspondence, the initial value is established.

An initially defined variable or array element may not be in blank common. A variable or array element in a labeled common block may be initially defined only in a block data subprogram.

7.2.3 FORMAT *Statement.* FORMAT statements are used in conjunction with the input/output of formatted records to provide conversion and editing information between the internal representation and the external character strings.

A FORMAT statement is of the form:

FORMAT $(q_1t_1z_1t_2z_2 \cdots z_{n-1}t_nq_2)$

where:

(1) $(q_1t_1z_1t_2z_2 \cdots z_{n-1}t_nq_2)$ is the format specification.

(2) Each q is a series of slashes or is empty.

(3) Each t is a field descriptor or group of field descriptors.

(4) Each z is a field separator.

(5) n may be zero.

A FORMAT statement must be labeled.

7.2.3.1 *Field Descriptors.* The format field descriptors are of the forms:

*sr*F*w.d*
*sr*E*w.d*
*sr*G*w.d*
*sr*D*w.d*
*r*I*w*
*r*L*w*
*r*A*w*
$n\text{H}h_1h_2 \cdots h_n$
*n*X

where:

(1) The letters F, E, G, D, I, L, A, H, and X indicate the manner of conversion and editing between the internal and external representations and are called the conversion codes.

(2) w and n are nonzero integer constants representing the width of the field in the external character string.

(3) d is an integer constant representing the number of digits in the fractional part of the external character string (except for G conversion code).

(4) r, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.

(5) s is optional and represents a scale factor designator.

(6) Each h is one of the characters capable of representation by the processor.

For all descriptors, the field width must be specified. For descriptors of the form *w.d* , the d must be specified, even if it is zero. Further, w must be greater than or equal to d.

The phrase *basic field descriptor* will be used to signify the field descriptor unmodified by s or r.

The internal representation of external fields will correspond to the internal representation of the corresponding type constants (4.2 and 5.1.1).

7.2.3.2 *Field Separators.* The format field separators are the slash and the comma. A series of slashes is also a field separator. The field descriptors or groups of field descriptors are separated by a field separator.

The slash is used not only to separate field descriptors, but to specify demarcation of formatted records. A formatted record is a string of characters. The lengths of the strings for a given external medium are dependent upon both the processor and the external medium.

The processing of the number of characters that can be contained in a record by an external medium does not of itself cause the introduction or inception of processing of the next record.

7.2.3.3 *Repeat Specifications.* Repetition of the field descriptors (except *n*H and *n*X) is accomplished by using the repeat count. If the input/output list warrants, the specified conversion will be interpreted repetitively up to the specified number of times.

Repetition of a group of field descriptors or field separators is accomplished by enclosing them within parentheses and optionally preceding the left parenthesis with an integer constant called the group repeat count indicating the number of times to interpret the enclosed grouping. If no group repeat count is specified, a group repeat count of one is assumed. This form of grouping is called a basic group.

A further grouping may be formed by enclosing field descriptors, field separators, or basic groups within parentheses. Again, a group repeat count may be specified. The parentheses enclosing the format specification are not considered as group delineating parentheses.

7.2.3.4 *Format Control Interaction with an Input/Output List.* The inception of execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided respectively by the next element of the input/output list, if one exists, and the next field descriptor obtained from the format specification. If there is an input/output list, at least one field descriptor other than *n*H or *n*X must exist.

When a READ statement is executed under format control, one record is read when the format control is initiated, and thereafter additional records are read only as the format specification demands. Such action may not require more characters of a record than it contains.

When a WRITE statement is executed under format control, writing of a record occurs each time the format specification demands that a new record be started. Termination of format control causes writing of the current record.

Except for the effects of repeat counts, the format specification is interpreted from left to right.

To each I, F, E, G, D, A, or L basic descriptor interpreted in a format specification, there corresponds one element specified by the input/output list, except that a complex element requires the interpretation of two F, E, or G basic descriptors. To each H or X basic descriptor there is no corresponding element specified by the input/output list, and the format control communicates information directly with the record. Whenever a slash is encountered, the format specification demands that a new record start or the preceding record terminate. During a READ operation, any unprocessed characters of the current record will be skipped at the time of termination of format control or when a slash is encountered.

Whenever the format control encounters an I, F, E, G, D, A, or L basic descriptor in a format specification, it determines if there is a corresponding element specified by the input/output list. If there is such an element, it transmits appropriately converted information between the element and the record and proceeds. If there is no corresponding element, the format control terminates.

If, however, the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified, the format control demands a new record start and control reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification. Note, this action of itself has no effect on the scale factor.

7.2.3.5 *Scale Factor.* A scale factor designator is defined for use with the F, E, G, and D conversions and is of the form:

nP

where n, the scale factor, is an integer constant or minus followed by an integer constant.

When the format control is initiated, a scale factor of zero is established. Once a scale factor has been established, it applies to all subsequently interpreted F, E, G, and D field descriptors, until another scale factor is encountered, and then that scale factor is established.

7.2.3.5.1 *Scale Factor Effects.* The scale factor n affects the appropriate conversions in the following manner:

(1) For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

> externally represented number equals internally represented number times the quantity ten raised to the nth power.

(2) For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.

(3) For E and D output, the basic real constant part of the output quantity is multiplied by 10^n and the exponent is reduced by n.

(4) For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion. If the effective use of E conversion is required, the scale factor has the same effect as with E output.

7.2.3.6 *Numeric Conversions.* The numeric field descriptors I, F, E, G, and D are used to specify input/output of integer real, double precision, and complex data.

(1) With all numeric input conversions, leading blanks are not significant and other blanks are zero. Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) With the F, E, G, and D input conversions, a decimal point appearing in the input field overrides the decimal point specification supplied by the field descriptor.

(3) With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width, leading blanks will be inserted in the output field.

(4) With all output conversions, the external representation of a negative value must be signed; a positive value may be signed.

(5) The number of characters produced by an output conversion must not exceed the field width.

7.2.3.6.1 *Integer Conversion.* The numeric field descriptor Iw indicates that the external field occupies w positions as an integer. The value of the list item appears, or is to appear, internally as an integer datum.

In the external input field, the character string must be in the form of an integer constant or signed integer constant (5.1.1.1), except for the interpretation of blanks (7.2.3.6).

The external output field consists of blanks, if necessary, followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value converted to an integer constant.

7.2.3.6.2 *Real Conversions.* There are three conversions available for use with real data: F, E, and G.

The numeric field descriptor F$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The basic form of the external input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. The basic form may be followed by an exponent of one of the following forms:

(1) Signed integer constant.
(2) E followed by an integer constant.
(3) E followed by a signed integer constant.
(4) D followed by an integer constant.
(5) D followed by a signed integer constant.

An exponent containing D is equivalent to an exponent containing E.

The external output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by string of digits containing a decimal point representing the magnitude of the internal value, as modified by the established scale factor, rounded to d fractional digits.

The numeric field descriptor E$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The form of the external input field is the same as for the F conversion.

The standard form of the external output field for a scale factor of zero is[1]

$$\xi 0.x_1 \cdots x_d Y$$

[1] ξ signifies no character position or minus in that position.

where:

(1) $x_1 \cdots x_d$ are the d most significant rounded digits of the value of the data to be output.

(2) Y is of one of the forms:

$$E \pm y_1 y_2 \quad \text{or} \quad \pm y_1 y_2 y_3$$

and has the significance of a decimal exponent (an alternative for the plus in the first of these forms is the character blank).

(3) The digit 0 in the aforementioned standard form may optionally be replaced by no character position.

(4) Each y is a digit.

The scale factor n controls the decimal normalization between the number part and the exponent part such that:

(1) If $n \leqq 0$, there will be exactly $-n$ leading zeros and $d + n$ significant digits after the decimal point.

(2) If $n > 0$, there will be exactly n significant digits to the left of the decimal point and $d - n + 1$ to the right of the decimal point.

The numeric field descriptor Gw.d indicates that the external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real datum.

Input processing is the same as for the F conversion.

The method of representation in the external output string is a function of the magnitude of the real datum being converted. Let N be the magnitude of the internal datum. The following tabulation exhibits a correspondence between N and the equivalent method of conversion that will be effected:

Magnitude of Datum	*Equivalent Conversion Effected*
$0.1 \leqq N < 1$	F$(w-4)$.d, 4X
$1 \leqq N < 10$	F$(w-4)$.$(d-1)$, 4X
.	.
.	.
.	.
$10^{d-2} \leqq N < 10^{d-1}$	F$(w-4)$.1, 4X
$10^{d-1} \leqq N < 10^{d}$	F$(w-4)$.0, 4X
Otherwise	sEw.d

Note that the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of F conversion.

7.2.3.6.3 *Double Precision Conversion.* The numeric field descriptor Dw.d indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a double precision datum.

The basic form of the external input field is the same as for real conversions.

The external output field is the same as for the E conversion, except that the character D may replace the character E in the exponent.

7.2.3.6.4 *Complex Conversion.* Since a complex datum consists of a pair of separate real data, the conversion is specified by two successively interpreted real field descriptors. The first of these supplies the real part. The second supplies the imaginary part.

7.2.3.7 *Logical Conversion.* The logical field descriptor Lw indicates that the external field occupies w positions as a string of information as defined below. The list item appears, or is to appear, internally as a logical datum.

The external input field must consist of optional blanks followed by a T or F followed by optional characters, for true and false, respectively.

The external output field consists of $w - 1$ blanks followed by a T or F as the value of the internal datum is true or false, respectively.

7.2.3.8 *Hollerith Field Descriptor.* Hollerith information may be transmitted by means of two field descriptors, nH and Aw:

(1) The nH descriptor causes Hollerith information to be read into, or written from, the n characters (including blanks) following the nH descriptor in the format specification itself.

(2) The Aw descriptor causes w Hollerith characters to be read into, or written from, a specified list element.

Let g be the number of characters representable in a single storage unit (7.2.1.3.1). If the field width specified for A input is greater than or equal to g, the rightmost g characters will be taken from the external input field. If the field width is less than g, the w characters will appear left justified with $w - g$ trailing blanks in the internal representation.

If the field width specified for A output is greater than g, the external output field will consist of $w - g$ blanks, followed by the g characters from the internal representation. If the field width is less than or equal to g, the external output field will consist of the leftmost w characters from the internal representation.

7.2.3.9 *Blank Field Descriptor.* The field descriptor for blanks is nX . On input, n characters of the external input record are skipped. On output, n blanks are inserted in the external output record.

7.2.3.10 *Format Specification in Arrays.* Any of the formatted input/output statements may contain an array name in place of the reference to a FORMAT statement label. At the time an array is referenced in such a manner, the first part of the information contained in the array, taken in the natural order, must constitute a valid format specification. There is no requirement on the information contained in the array following the right parenthesis that ends the format specification.

The format specification which is to be inserted in the array has the same form as that defined for a FORMAT statement; that is, begins with a left parenthesis and ends with a right parenthesis. An nH field descriptor may not be part of a format specification within an array.

The format specification may be inserted in the array by use of a data initialization statement, or by use of a READ statement together with an A format.

8. PROCEDURES AND SUBPROGRAMS

There are four categories of procedures: statement functions, intrinsic functions, external functions, and external subroutines. The first three categories are referred to collectively as functions or function procedures; the last as subroutines or subroutine procedures. There are two categories of subprograms: procedure subprograms and specification subprograms. Function subprograms and subroutine subprograms are classified as procedure subprograms. Block data subprograms are classified as specification subprograms. Type rules for function procedures are given in 5.3.

8.1 Statement Functions. A statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic or logical assignment statement.

In a given program unit, all statement function definitions must precede the first executable statement of the program unit and must follow the specification statements, if any. The name of a statement function must not appear in an EXTERNAL statement, nor as a variable name or an array name in the same program unit.

8.1.1 *Defining Statement Functions.* A statement function is defined by a statement of the form:

$$f(a_1, a_2, \cdots, a_n) = e$$

where f is the function name, e is an expression, and the relationship between f and e must conform to the assignment rules in 7.1.1.1 and 7.1.1.2. The a's are distinct variable names, called the *dummy arguments* of the function. Since these are dummy arguments, their names, which serve only to indicate type, number, and order of arguments, may be the same as variable names of the same type appearing elsewhere in the program unit.

Aside from the dummy arguments, the expression e may only contain:

(1) Non-Hollerith constants.

(2) Variable references.

(3) Intrinsic function references.

(4) References to previously defined statement functions.

(5) External function references.

8.1.2 *Referencing Statement Functions.* A statement function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

Execution of a statement function reference results in an association (10.2.2) of actual argument values with the corresponding dummy arguments in the expression of the function definition, and an evaluation of the expression. Following this, the resultant value is made available to the expression that contained the function reference.

8.2 Intrinsic Functions and Their Reference. The symbolic names of the intrinsic functions (see Table 3) are predefined to the processor and have a special meaning and type if the name satisfies the conditions of 10.1.7.

An intrinsic function is referenced by using its reference as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Table 3 and may be any expression of the specified type. The intrinsic functions AMOD, MOD, SIGN, ISIGN, and DSIGN are not defined when the value of the second argument is zero.

Execution of an intrinsic function reference results in the actions specified in Table 3 based on the values of the actual arguments. Following this, the resultant value is made available to the expression that contained the function reference.

8.3 External Functions. An external function is defined externally to the program unit that references it. An external function defined by Fortran statements headed by a FUNCTION statement is called a function subprogram.

8.3.1 *Defining Function Subprograms.* A FUNCTION statement is of the form:

$$t \text{ FUNCTION } f\ (a_1, a_2, \cdots, a_n)$$

where:

(1) t is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, or is empty.

(2) f is the symbolic name of the function to be defined.

(3) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

TABLE 3. Intrinsic Functions

Intrinsic Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of:* Argument	*Type of:* Function
Absolute Value	$\lvert a \rvert$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
Truncation	Sign of a times largest integer $\leq \lvert a \rvert$	1	AINT	Real	Real
			INT	Real	Integer
			IDINT	Double	Integer
Remaindering* (see note below)	$a_1 \pmod{a_2}$	2	AMOD	Real	Real
			MOD	Integer	Integer
Choosing Largest Value	Max $(a_1, a_2, \cdots)$	$\geqq 2$	AMAX0	Integer	Real
			AMAX1	Real	Real
			MAX0	Integer	Integer
			MAX1	Real	Integer
			DMAX1	Double	Double
Choosing Smallest Value	Min $(a_1, a_2, \cdots)$	$\geqq 2$	AMIN0	Integer	Real
			AMIN1	Real	Real
			MIN0	Integer	Integer
			MIN1	Real	Integer
			DMIN1	Double	Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $\lvert a_1 \rvert$	2	SIGN	Real	Real
			ISIGN	Integer	Integer
			DSIGN	Double	Double
Positive Difference	$a_1 -$ Min (a_1, a_2)	2	DIM	Real	Real
			IDIM	Integer	Integer
Obtain Most Significant Part of Double Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double
Express Two Real Arguments in Complex Form	$a_1 + a_2\sqrt{-1}$	2	CMPLX	Real	Complex
Obtain Conjugate of a Complex Argument		1	CONJG	Complex	Complex

*The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

Function subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the function must also appear as a variable name in the defining subprogram. During every execution of the subprogram, this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in this subprogram is called the value of the function.

(2) The symbolic name of the function must not appear in any nonexecutable statement in this program unit, except as the symbolic name of the function in the FUNCTION statement.

(3) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram.

(4) The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

(5) The function subprogram may contain any statements except BLOCK DATA, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.

(6) The function subprogram must contain at least one RETURN statement.

8.3.2 *Referencing External Functions.* An external function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program unit. An actual argument in an external function reference may be one of the following:

(1) A variable name.
(2) An array element name.
(3) An array name.
(4) Any other expression.
(5) The name of an external procedure.

If an actual argument is an external function name or a subroutine name, then the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name. Execution of an external function reference as described in the foregoing, results in an association (10.2.2) of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (4) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken. An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a function reference causes a dummy argument in the referenced function to become associated with another dummy argument in the same function or with an entity in common, a definition of either within the function is prohibited.

Unless it is a dummy argument, an external function is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.3.3 *Basic External Functions.* FORTRAN processors must supply the external functions listed in Table 4. Referencing of these functions is accomplished as described in (8.3.2). Arguments for which the result of these functions is not mathematically defined or is of type other than that specified are improper.

8.4 SUBROUTINE. An external subroutine is defined externally to the program unit that references it. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a subroutine subprogram.

TABLE 4. BASIC EXTERNAL FUNCTIONS

Basic External Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of: Argument*	*Function*
Exponential	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
		1	CEXP	Complex	Complex
Natural Logarithm	$\log_e (a)$	1	ALOG	Real	Real
		1	DLOG	Double	Double
		1	CLOG	Complex	Complex
Common Logarithm	$\log_{10} (a)$	1	ALOG10	Real	Real
			DLOG10	Double	Double
Trigonometric Sine	$\sin (a)$	1	SIN	Real	Real
		1	DSIN	Double	Double
		1	CSIN	Complex	Complex
Trigonometric Cosine	$\cos (a)$	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
Hyperbolic Tangent	$\tanh (a)$	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
		1	CSQRT	Complex	Complex
Arctangent	$\arctan (a)$	1	ATAN	Real	Real
		1	DATAN	Double	Double
	$\arctan (a_1/a_2)$	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Remaindering*	$a_1 \pmod{a_2}$	2	DMOD	Double	Double
Modulus		1	CABS	Complex	Real

*The function DMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

8.4.1 *Defining Subroutine Subprograms.* A SUBROUTINE statement is of one of the forms:

SUBROUTINE $s\ (a_1, a_2, \cdots, a_n)$

or

SUBROUTINE s

where:

(1) s is the symbolic name of the subroutine to be defined.

(2) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

Subroutine subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the subroutine must not appear in any statement in this subprogram except as the symbolic name of the subroutine in the SUBROUTINE statement itself.

(2) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram.

(3) The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.

(4) The subroutine subprogram may contain any statements except BLOCK DATA, FUNCTION, another SUB-

ROUTINE statement, or any statement that directly or indirectly references the subroutine being defined.

(5) The subroutine subprogram must contain at least one RETURN statement.

8.4.2 *Referencing Subroutines.* A subroutine is referenced by a CALL statement (7.1.2.4). The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program. The use of a Hollerith constant as an actual argument is an exception to the rule requiring agreement of type. An actual argument in a subroutine reference may be one of the following:

(1) A Hollerith constant.
(2) A variable name.
(3) An array element name.
(4) An array name.
(5) Any other expression.
(6) The name of an external procedure.

If an actual argument is an external function name or a subroutine name, the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name.

Execution of a subroutine reference as described in the foregoing results in an association of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (5) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in common, a definition of either entity within the subroutine is prohibited.

Unless it is a dummy argument, a subroutine is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.5 Block Data Subprogram. A BLOCK DATA statement is of the form:

BLOCK DATA

This statement may only appear as the first statement of specification subprograms that are called block data subprograms, and that are used to enter initial values into elements of labeled common blocks. This special subprogram contains only type-statements, EQUIVALENCE, DATA, DIMENSION, and COMMON statements.

If any entity of a given common block is being given an initial value in such a subprogram, a complete set of specification statements for the entire block must be included, even though some of the elements of the block do not appear in DATA statements. Initial values may be entered into more than one block in a single subprogram.

9. PROGRAMS

An executable program is a collection of statements, comment lines, and end lines that completely (except for input data values and their effects) describe a computing procedure.

9.1 Program Components. Programs consist of program parts, program bodies, and subprogram statements.

9.1.1 *Program Part.* A program part must contain at least one executable statement and may contain FORMAT statements, and data initialization statements. It need not contain any statements from either of the latter two classes of statement. This collection of statements may optionally be preceded by statement function definitions, data initialization statements, and FORMAT statements. As before only some or none of these need be present.

9.1.2 *Program Body.* A program body is a collection of specification statements, FORMAT statements or both, or neither, followed by a program part, followed by an end line.

9.1.3 *Subprogram.* A subprogram consists of a SUBROUTINE or FUNCTION statement followed by a program body, or is a block data subprogram.

9.1.4 *Block Data Subprogram.* A block data subprogram consists of a BLOCK DATA statement, followed by the appropriate (8.5) specification statements, followed by data initialization statements, followed by an end line.

9.1.5 *Main Program.* A main program consists of a program body.

9.1.6 *Executable Program.* An executable program consists of a main program plus any number of subprograms, external procedures, or both.

9.1.7 *Program Unit.* A program unit is a main program or a subprogram.

9.2 Normal Execution Sequence. When an executable program begins operation, execution commences with the execution of the first executable statement of the main program. A subprogram, when referenced, starts execution with execution of the first executable statement of that subprogram. Unless a statement is a GO TO, arithmetic IF, RETURN, or STOP statement or the terminal statement of a DO, completion of execution of that statement causes execution of the next following executable statement. The sequence of execution following execution of any of these statements is described in Section 7. A program part may not (in the sense of 1.1) contain an executable statement that can never be executed.

A program part must contain a first executable statement.

10. INTRA- AND INTERPROGRAM RELATIONSHIPS*

10.1 Symbolic Names. A symbolic name has been defined to consist of from one to six alphanumeric characters, the first of which must be alphabetic. Sequences of characters that are format field descriptors or uniquely identify certain statement types, e.g., GO TO, READ, FORMAT, etc. are not symbolic names in such occurrences nor do they form the first characters of symbolic names in these cases. In a program unit, a symbolic name (perhaps qualified by a subscript) must identify an element of one (and usually only one) of the following classes:

Class I	An array and the elements of that array.
Class II	A variable.
Class III	A statement function.
Class IV	An intrinsic function.
Class V	An external function.
Class VI	A subroutine.

Class VII An external procedure which cannot be classified as either a subroutine or an external function in the program unit in question.

Class VIII A block name.

10.1.1 *Restrictions on Class.* A symbolic name in Class VIII in a program unit may also be in any one of the Classes I, II, or III in that program unit.

In the program unit in which a symbolic name in Class V appears immediately following the word FUNCTION in a FUNCTION statement, that name must also be in Class II.

Once a symbolic name is used in Class V, VI, VII, or VIII in any unit of an executable program, no other program unit of that executable program may use that name to identify an entity of these classes other than the one originally identified. In the totality of the program units that make up an executable program, a Class VII name must be associated with a Class V or VI name. Class VII can only exist locally in program units.

In a program unit, no symbolic name can be in more than one class except as noted in the foregoing. There are no restrictions on uses of symbolic names in different program units of an executable program other than those noted in the foregoing.

10.1.2 *Implications of Mentions in Specification and DATA Statements.* A symbolic name is in Class I if and only if it appears as a declarator name. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a COMMON statement (other than as a block name) is either in Class I, or in Class II but not Class V. (8.3.1) Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EQUIVALENCE statement is either in Class I, or in Class II but not Class V. (8.3.1).

A symbolic name that appears in a type-statement cannot be in Class VI or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EXTERNAL statement is in either Class V, Class VI, or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a DATA statement is in either Class I, or in Class II but not Class V. (8.3.1) In an executable program, a storage unit (7.2.1.3.1) may have its value initialized one time at the most.

10.1.3 *Array and Array Element.* In a program unit, any appearance of a symbolic name that identifies an array must be immediately followed by a subscript, except for the following cases:

(1) In the list of an input/output statement.

(2) In a list of dummy arguments.

(3) In the list of actual arguments in a reference to an external procedure.

(4) In a COMMON statement.

(5) In a type-statement.

Only when an actual argument of an external procedure reference is an array name or an array element name may the corresponding dummy argument be an array name. If the actual argument is an array name, the length of the dummy argument array must be no greater than the length of the actual argument array. If the actual argument is an array element name, the length of the dummy argument array must be less than or equal to the length of the actual argument array plus one minus the value of the subscript of the array element.

10.1.4 *External Procedures.* The only case when a symbolic name is in Class VII occurs when that name appears only in an EXTERNAL statement and as an actual argument to an external procedure in a program unit.

Only when an actual argument of an external procedure reference is an external procedure name may the corresponding dummy argument be an external procedure name.

In the execution of an executable program, a procedure subprogram may not be referenced twice without the execution of a RETURN statement in that procedure having intervened.

10.1.5 *Subroutine.* A symbolic name is in Class VI if it appears:

(1) Immediately following the word SUBROUTINE in a SUBROUTINE statement.

(2) Immediately following the word CALL in a CALL statement.

10.1.6 *Statement Function.* A symbolic name is in Class III in a program unit if and only if it meets all three of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in Class I.

(2) Every appearance of the name, except in a type-statement, is immediately followed by a left parenthesis.

(3) A function defining statement (8.1.1) is present for that symbolic name.

10.1.7 *Intrinsic Function.* A symbolic name is in Class IV in a program unit if and only if it meets all four of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in Class I or Class III.

(2) The symbolic name appears in the name column of the table in Section 8.2.

(3) The symbolic name does not appear in a type-statement of type different from the intrinsic type specified in the table.

(4) Every appearance of the symbolic name (except in a type-statement as described in the foregoing) is immediately followed by an actual argument list enclosed in parentheses.

The use of an intrinsic function in a program unit of an executable program does not preclude the use of the same symbolic name to identify some other entity in a different program unit of that executable program.

10.1.8 *External Function.* A symbolic name is in Class V if it:

(1) Appears immediately following the word FUNCTION in a FUNCTION statement

(2) Is not in Class I, Class III, Class IV, or Class VI and appears immediately followed by a left parenthesis on every occurrence except in a type-statement, in an EXTERNAL statement, or as an actual argument. There must be at least one such appearance in the program unit in which it is so used.

10.1.9 *Variable.* In a program unit, a symbolic name is in Class II if it meets all three of the following conditions:

(1) It is not in Class VI or Class VII.

(2) It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement.

(3) It occurs other than in a Class VIII appearance.

10.1.10 *Block Name.* A symbolic name is in Class VIII if and only if it is used as a block name in a COMMON statement.

10.2 DEFINITION. There are two levels of definition of numeric values, first level definition and second level definition. The concept of definition on the first level applies to array elements and variables; that of second level definition to integer variables only. These concepts are defined in terms of progression of execution; and thus, an executable program, complete and in execution, is assumed in what follows.

There are two other varieties of definition that should be noted. The first, effected by GO TO assignment and referring to an integer variable being defined with other than an integer value, is discussed in 7.1.1.3 and 7.1.2.1.2; the second, which refers to when an external procedure may be referenced, will be discussed in the next section.

In what follows, otherwise unqualified use of the terms definition and undefinition (or their alternate forms) as applied to variables and array elements will imply modification by the phrase on the first level.

10.2.1 *Definition of Procedures.* If an executable program contains information describing an external procedure, such an external procedure with the applicable symbolic name is defined for use in that executable program. An external function reference or subroutine reference (as the case may be) to that symbolic name may then appear in the executable program, provided that number of arguments agrees between definition and reference. In addition, for an external function, the type of function must agree between definition and reference. Other restrictions on agreements are contained in 8.3.1., 8.3.2, 8.4.1., 8.4.2., 10.1.3, and 10.1.4.

The basic external functions listed in (8.3.3) are always defined and may be referenced subject to the restrictions alluded to in the foregoing.

A symbolic name in Class III or Class IV is defined for such use.

10.2.2 *Associations That Effect Definition.* Entities may become associated by:

(1) COMMON association.

(2) EQUIVALENCE association.

(3) Argument substitution.

Multiple association to one or more entities can be the result of combinations of the foregoing. Any definition or undefinition of one of a set of associated entities effects the definition or undefinition of each entity of the entire set.

For purposes of definition, in a program unit there is no association between any two entities both of which appear in COMMON statements. Further, there is no other association for common and equivalenced entities other than those stated in 7.2.1.3.1 and 7.2.1.4.

If an actual argument of an external procedure reference is an array name, an array element name, or a variable name, then the discussions in 10.1.3 and 10.2.1 allow an association of dummy arguments with the actual arguments only between the time of execution of the first executable statement of the procedure and the inception of execution of the next encountered RETURN statement of that procedure. Note specifically that this association can be carried through more than one level of external procedure reference.

In what follows, variables or array elements associated by the information in 7.2.1.3.1 and 7.2.1.4 will be equivalent if and only if they are of the same type.

If an entity of a given type becomes defined, then all associated entities of different type become undefined at the same time, while all associated entities of the same type become defined unless otherwise noted.

Association by argument substitution is only valid in the case of identity of type, so the rule in this case is that an entity created by argument substitution is defined at time of entry if and only if the actual argument was defined. If an entity created by argument substitution becomes defined or undefined (while the association exists) during execution of a subprogram, then the corresponding actual entities in all calling program units becomes defined or undefined accordingly.

10.2.3 *Events That Effect Definition.* Variables and array elements become initially defined if and only if their names are associated in a data initialization statement with a constant of the same type as the variable or array in question. Any entity not initially defined is undefined at the time of the first execution of the first executable statement of the main program. Redefinition of a defined entity is always permissible except for certain integer variables (7.1.2.8, 7.1.3.1.1, and 7.2.1.1.2) or certain entities in subprograms (6.4, 8.3.2, and 8.4.2).

Variables and array elements become defined or redefined as follows:

(1) Completion of execution of an arithmetic or logical assignment statement causes definition of the entity that precedes the equals.

(2) As execution of an input statement proceeds, each entity, which is assigned a value of its corresponding type from the input medium, is defined at the time of such association. Only at the completion of execution of the statement do associated entities of the same type become defined.

(3) Completion of execution of a DO statement causes definition of the control variable.

(4) Inception of execution of action specified by a DO-implied list causes definition of the control variable.

Variables and array elements become undefined as follows:

(1) At the time a DO is satisfied, the control variable becomes undefined.

(2) Completion of execution of an ASSIGN statement causes undefinition of the integer variable in the statement.

(3) Certain entities in function subprograms (10.2.9) become undefined.

(4) Completion of execution of action specified by a DO-implied list causes undefinition of the control variable.

(5) When an associated entity of different type becomes defined.

(6) When an associated entity of the same type becomes undefined.

10.2.4 *Entities in Blank Common.* Entities in blank common and those entities associated with them may not be initially defined.

Such entities, once defined by any of the rules previously mentioned, remain defined until they become undefined.

10.2.5 *Entities in Labeled Common.* Entities in labeled common or any associates of those entities may be initially defined.

A program unit contains a labeled common block name if the name appears as a block name in the program unit. If a main program or referenced subprogram contains a labeled common block name, any entity in the block (and its associates) once defined remain defined until they become undefined.

It should be noted that redefinition of an initially defined entity will allow later undefinition of that entity.

Specifically, if a subprogram contains a labeled common block name that is not contained in any program unit currently referencing the subprogram directly or indirectly, the execution of a RETURN statement in the subprogram causes undefinition of all entities in the block (and their associates) except for initially defined entities that have maintained their initial definitions.

10.2.6 *Entities Not in Common.* An entity not in common except for a dummy argument or the value of a function may be initially defined.

Such entities once defined by any of the rules previously mentioned, remain defined until they become undefined.

If such an entity is in a subprogram, the completion of execution of a RETURN statement in that subprogram causes all such entities and their associates at that time (except for initially defined entities that have not been redefined or become undefined) to become undefined. In this respect, it should be noted that the association between dummy arguments and actual arguments is terminated at the inception of execution of the RETURN statement.

Again, it should be emphasized, the redefinition of an initially defined entity can result in a subsequent undefinition of that entity.

10.2.7 *Basic Block.* In a program unit, a basic block is a group of one or more executable statements defined as follows.

The following statements are block terminal statements:

(1) DO statement.

(2) CALL statement.

(3) GO TO statement of all types.

(4) Arithmetic IF statement.

(5) STOP statement.

(6) RETURN statement.

(7) The first executable statement, if it exists, preceding a statement whose label is mentioned in a GO TO or arithmetic IF statement.

(8) An arithmetic statement in which an integer variable precedes the equals.

(9) A READ statement with an integer variable in the list.

(10) A logical IF containing any of the admissible forms given in the foregoing.

The following statements are block initial statements:

(1) The first executable statement of a program unit.

(2) The first executable statement, if it exists, following a block terminal statement.

Every block initial statement defines a basic block. If that initial statement is also a block terminal statement, the basic block consists of that one statement. Otherwise, the basic block consists of the initial statement and all executable statements that follow until a block terminal statement is encountered. The terminal statement is included in the basic block.

10.2.7.1 *Last Executable Statement.* In a program unit the last executable statement (which cannot be part of a logical IF) must be one of the following statements: GO TO statement, arithmetic IF statement, STOP statement, or RETURN statement.

10.2.8 *Second Level Definition.* Integer variables must be defined on the second level when used in subscripts and computed GO TO statements.

Redefinition of an integer entity causes all associated variables to be undefined for use on the second level during this execution of this program unit until the associated integer variable is explicitly redefined.

Except as just noted, an integer variable is defined on the second level upon execution of the initial statement of a basic block only if both of the following conditions apply:

(1) The variable is used in a subscript or in a computed GO TO in the basic block in question.

(2) The variable is defined on the first level at the time of execution of the initial statement in question.

This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block in question.

(2) The variable in question becomes undefined or receives a new definition on the first level.

At this time, the variable becomes undefined on the second level.

In addition, the occurrence of an integer variable in the list of an input statement in which that integer variable appears following in a subscript causes that variable to be defined on the second level. This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block containing the input statement.

(2) The variable becomes undefined or receives a new definition on the first level.

An integer variable defined as the control variable of a DO-implied list is defined on the second level over the range of that DO-implied list and only over that range.

10.2.9 *Certain Entities in Function Subprograms.* If a function subprogram is referenced more than once with an identical argument list in a single statement, the execution of that subprogram must yield identical results for those cases mentioned, no matter what the order of evaluation of the statement.

If a statement contains a factor that may not be evaluated (6.4), and if this factor contains a function reference, then all entities that might be defined in that reference become undefined at the completion of evaluation of the expression containing the factor.

10.3 Definition Requirements for Use of Entities. Any variable referenced in a subscript or a computed GO TO must be defined on the second level at the time of this use.

Any variable, array element, or function referenced as a primary in an expression and any subroutine referenced by a CALL statement must be defined at the time of this use. In the case where an actual argument in the argument list of an external procedure reference is a variable name or an array element name, this in itself is not a requirement that the entity be defined at the time of the procedure reference; however, when such an argument is an external procedure name, it must be defined.

Any variable used as an initial value, terminal value, or incrementation value of a DO statement or a DO-implied list must be defined at the time of this use.

Any variable used to identify an input/output unit must be defined at the time of this use.

At the time of execution of a RETURN statement in a function subprogram, the value (8.3.1) of that function must be defined.

At the time of execution of an output statement, every entity whose value is to be transferred to the output medium must be defined unless the output is under control of a format specification and the corresponding conversion code is A. If the output is under control of a format specification, a correct association of conversion code with type of entity is required unless the conversion code is A The following are the correct associations: I with integer; D with double precision; E, F, and G with real and complex; and L with logical.

INDEX

HONEYWELL INFORMATION SYSTEMS

Publications Remarks Form*

TITLE:	SERIES 16 FORTRAN IV	ORDER No.:	BX32, REV. 0
		DATED:	APRIL 1967

ERRORS IN PUBLICATION:

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION:

(Please Print)

FROM: NAME ______________________ DATE: ______________

COMPANY ______________________

TITLE ______________________

____

____

*Your comments will be promptly investigated by appropriate technical personnel, action will be taken as required, and you will receive a written reply. If you do not require a written reply, please check here. ☐

ONG LINE

CUT ALONG

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MASS. 02181

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

POSTAGE WILL BE PAID BY:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MASS. 02181

ATTN: PUBLICATIONS, MS 050

Honeywell